Universidad Carlos III de Madrid
School of Engineering
Computer Science and Engineering Department
Computer Architecture and Technology Area

Ph.D. Thesis

New Approaches to Data Access in Large-Scale Distributed Systems

Author: Borja Bergua Guerra
Advisors: Prof. Dr. Félix García Carballeira
Dr. Alejandro Calderón Mateos

Leganés, Madrid
December, 2015

TESIS DOCTORAL

New Approaches to Data Access in Large-Scale Distributed Systems

AUTOR:		**Borja Bergua Guerra**
DIRECTORES:	**Prof. Dr.**	**Félix García Carballeira**
	Dr.	**Alejandro Calderón Mateos**

FIRMA DEL TRIBUNAL CALIFICADOR

FIRMA

PRESIDENTE:

VOCAL:

SECRETARIO:

CALIFICACIÓN:

Leganés, a 21 de Diciembre de 2015

Acknowledgments

Many people appeared in my life during this thesis. Many people have contributed to making this thesis possible. My sincere gratitude to all of you.

First and foremost, I would like to thank specially the incredible support of my advisor Dr. Félix García Carballeira. It is impossible to describe how much I owe you. I have always found good advises and wise experience in you. You have been the most relevant person in my professional life. No doubt. You know how sad it was for me to stop working with you. But life must go on. And I wish to be, someday, as important for a team mate as you were for me. Thank you very much Félix.

I would also like to thank my co-advisor Dr. Alejandro Calderón. Since a I met you, you have always been a good friend. And very probably, we will be for life.

I would like to acknowledge Dr. Jesús Carretero, head of ARCOS research group of Universidad Carlos III de Madrid. Thank you very much for your patience and confidence.

Very special thanks go to Javi "Zor", Luismi, Alejandra, Maria Cristina, and "Fortran". You have been much more than simply team mates.

I would also like to thank all the people that are or have been members of ARCOS team during all these years: Javi "Doc", José Daniel, Florin, David, Paco, Sole, Fernando, Rosa, J.C. Pichel, Daniel, Juanma, Alberto "Garci", Carlos, Gonzalo, Pablo, Gabriel, Fran "Duro", Ernesto, JuanFra, José Luis G. Compean, J. Manuel P. Lobato, M. Gregoria, M. Blanca, Chema and Marga.

I made very good friends in the admin lab of the Computer Science department. Óscar, the very first person I worked with, here at university, back in 2003. And Roberto, my coffee mate and confidant. How many good moments both of you gave me. Thank you so much. Impossible to remember everyone, here are some of them: Jaime, Rafa, Fran, Álvaro, Adrian, Carlos, Iván, ...

As part of the international PhD, I did a research stay at Laboratoire d'Informatique de Grenoble. First of all, I would like to thank specially Dr. Derrick Kondo, for hosting my visit, and for all his support to me. I would also like to thank many people there at Grenoble that made me feel like at home: Arnaud Legrand, Pierre Navarro, Laurent Bobelin, Corinne Touati, Cristian Ruiz, Christophe Laferrière, Joseph Emeras, Lucas Schnorr, Eric Amat, Rodrigue Chakode, and many others (sorry, my memory is fragile). Thanks so much to all of you. You have a friend in Spain.

Also, during my time in Grenoble, I had the pleasure to meet a fabulous group of friends: Laura, Ana, Irene, Sonia, ... Every time I see a *Sephora* I put a smile on my face.

The last part of this thesis was finished while working at Pragsis Technologies S.L. It is being an exciting stage in my professional life, surfing the wave of Big Data & Data Science. First of all, I would like to thank Pedro Agudo, for giving me this opportunity and all the help that I needed to finish this thesis. To David Millán for believing in me for this job. And very specially to Daniel Palomar, my perfect complement and good friend. Also, I would like to thank all the good friends I have made at Pragsis: Ramiro, Miguel, Gerardo, Dima, Aitor, Sole, Fernando, Marta, Benjamin, José David, Fernanda, Verónica, Raúl, Rubén, Óliver, David Santibáñez, Jaime, José Manuel, Santiago, Antonio, José María, Belén, Mónica, Eduardo, Alberto, and many others.

Finally, I would like to thank my closest family: parents, brother, mother's aunt, and very specially to Nuria, my friend, my soul mate, my love, my everything. Only you know how much this thesis has stolen to us. And only you know how much I owe you. Finishing this thesis does not pay for all the lost moments, but this is the end of a road, and a new and exciting road will born in April. That will surely compensate everything.

Borja Bergua.

Abstract

A great number of scientific projects need supercomputing resources, such as, for example, those carried out in physics, astrophysics, chemistry, pharmacology, etc. Most of them generate, as well, a great amount of data; for example, a some minutes long experiment in a particle accelerator generates several terabytes of data.

In the last years, high-performance computing environments have evolved towards large-scale distributed systems such as Grids, Clouds, and Volunteer Computing environments. Managing a great volume of data in these environments means an added huge problem since the data have to travel from one site to another through the internet.

In this work a novel generic I/O architecture for large-scale distributed systems used for high-performance and high-throughput computing will be proposed. This solution is based on applying parallel I/O techniques to remote data access. Novel replication and data search schemes will also be proposed; schemes that, combined with the above techniques, will allow to improve the performance of those applications that execute in these environments. In addition, it will be proposed to develop simulation tools that allow to test these and other ideas without needing to use real platforms due to their technical and logistic limitations. An initial prototype of this solution has been evaluated and the results show a noteworthy improvement regarding to data access compared to existing solutions.

Resumen

Un gran número de proyectos científicos necesitan recursos de supercomputación como, por ejemplo, los llevados a cabo en física, astrofísica, química, farmacologia, etc. Muchos de ellos generan, además, una gran cantidad de datos; por ejemplo, un experimento de unos minutos de duración en un acelerador de partículas genera varios terabytes de datos.

Los entornos de computación de altas prestaciones han evolucionado en los últimos años hacia sistemas distribuidos a gran escala tales como Grids, Clouds y entornos de computación voluntaria. En estos entornos gestionar un gran volumen de datos supone un problema añadido de importantes dimensiones ya que los datos tienen que viajar de un sitio a otro a través de internet.

En este trabajo se propondrá una nueva arquitectura de E/S genérica para sistemas distribuidos a gran escala usados para cómputo de altas prestaciones y de alta productividad. Esta solución se basa en la aplicación de técnicas de E/S paralela al acceso remoto a los datos. Así mismo, se estudiarán y propondrán nuevos esquemas de replicación y búsqueda de datos que, en combinación con las técnicas anteriores, permitan mejorar las prestaciones de aquellas aplicaciones que ejecuten en este tipo de entornos. También se propone desarrollar herramientas de simulación que permitan probar estas y otras ideas sin necesidad de recurrir a una plataforma real debido a las limitaciones técnicas y logísticas que ello supone. Se ha evaluado un prototipo inicial de esta solución y los resultados muestran una mejora significativa en el acceso a los datos sobre las soluciones existentes.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In the last years the computing power of high-performance systems has continued increasing at an exponential rate, making even more challenging the access to large data sets. The ever increasing gap between I/O subsystems and processor speeds has driven researchers to look for scalable I/O solutions, including parallel file systems and I/O libraries.

A typical parallel file system stripes the file data and metadata over several independent disks managed by I/O nodes in order to allow parallel file access from several compute nodes. Examples of popular file systems include GPFS [Schmuck and Haskin (2002)], PVFS [Ligon and Ross (1999)] and Lustre [Cluster File Systems Inc. (2002)]. These parallel file systems manage the storage of several clusters and supercomputers from the top 500 list.

In distributed systems there are file systems that offer parallel access using multiple replicas. Examples of such distributed file systems include CEPH [Weil et al. (2006a)], GlusterFS [Red Hat, Inc (2015)] or HDFS [The Apache Software Foundation (2015a)]. However these file systems require deploying specific services, which does not make them suitable for working completely at client-side. And, also, do not address specifically the problem of optimizing replica selection for small accesses in addition to big transfers.

1.1 Motivation

The parallel I/O solutions proposed so far in literature address either clusters of computers or supercomputers. Given the proprietary nature of many supercomputers, the majority of works had concentrated on clusters of computers. Only a limited number of papers have proposed novel solutions for scalable parallel I/O systems in large-scale distributed systems. Nevertheless, distributed systems have a complex architecture consisting of several networks and tiers (computing, I/O, storage), and, consequently, a potential parallel access to data. This type of architecture provides a rich set of opportunities for parallel I/O optimizations. Existing approaches concentrate on large sequential movements of data across networks, or the use of replicas to select the closest sites.

Data intensive applications need to access to great amounts of data. The I/O access patterns of

scientific parallel applications often consist of accesses to a large number of small, non-contiguous pieces of data. Furthermore, many current data access libraries such as HDF5 and NetCDF rely heavily on small data accesses to store individual data elements in a common large file [The HDF group (2012), Li et al. (2003)]. For small file accesses the performance is dominated by the latency of network transfers and disks. Additionally, parallel scientific applications lead to interleaved file access patterns with high interprocess spatial locality at the I/0 nodes [Nieuwejaar et al. (1996), Simitici and Reed (1998)]. For big file accesses the performance is dominated by the bandwidth of networks. These characteristics of the access patterns of data intensive applications motivated several researchers to propose the use of replication schemes to improve data access and availability [Amjad et al. (2012)].

However, these optimizations do not benefit of I/O techniques used in clusters or supercomputers. In this thesis, we propose to demonstrate that parallel and remote techniques to data access, as those typically used in clusters, results in a significant performance improvements, scalability, and better resource usage in large-scale distributed systems.

1.2 Objectives

The main objectives of this dissertation are:

- To propose a **generic I/O middleware architecture** for large-scale distributed systems.
- To design a **replica selection algorithm** for configuring access virtual parallel partitions for large-scale distributed systems.

By fulfilling the above objectives we try to obtain the following benefits:

- **Generic architecture**. This architecture targets large-scale distributed systems. The majority of existing approaches target cluster and supercomputer architectures, due to the high-performance nature of clusters and supercomputers. The novelty of this dissertation consists in proposing a generic architecture based on open-source software, encompassing large-scale distributed systems.
- **Portability**. Portability is achieved by using well known technologies like HTTP and GridFTP, de-facto transfer standards in distributed architectures.
- **Scalability**. This architecture should be scalable to systems with thousands of machines. The novelty of our approach consists in addressing scalability by using multiple replicas of the data in parallel.
- **High-throughput**. The application should be offered high-throughput parallel and remote I/O. High throughput is obtained from parallel and remote access to independent storage resources, a tight integration between the applications and the middleware, and overlap of computation, communication, and I/O.
- **High resources utilization**. The middleware should achieve a high utilization of available resources such as storage and networks.

- **Transparency and simplicity of use**. This architecture can be used transparently by the user. Optionally, the user may chose different parallel I/O optimizations in a straightforward way.

1.3 Structure and contents

The remainder of this document is structured in the following way:

- Chapter 2 *State of the art* contains the state of the art.
- Chapter 3 *Proposal of a generic I/O arch. for large-scale distributed systems* presents the architecture of the parallel file system proposed to solve some of the I/O problems that can be found in large-scale distributed systems.
- Chapter 4 *Evaluation of the parallel FS in Grid and volunteer comp. envs* reports performance results for both grid and volunteer computing environments.
- Chapter 5 *Optimizations for replica selection in large-scale distributed systems* presents optimizactions designed towards improving data access in large-scale distributed environments, and reports performance results of the optimizations for Grid and volunteer computing environments.
- Chapter 6 *Conclusions and future work* contains a summary of this thesis, publications, and future plans.
- Appendix A *Architecture, design, and implementation of Expand* contains an detailed study of the *Expand* parallel file system.

Chapter 2

State of the art

This chapter presents the state of the art related to this dissertation and the background concepts necessary for the understanding of the solution. The material is organized in six sections: Supercomputing, Large-scale distributed systems, Data-intensive computing paradigms, Data transport technologies in large-scale distributed systems, Data replication, and Distributed storage.

2.1 Supercomputing

A supercomputer is a computer at the frontline of current processing capacity, particularly speed of calculation. Supercomputers were introduced in the 1960s and were designed primarily by Seymour Cray at Control Data Corporation (CDC), and later at Cray Research. While the supercomputers of the 1970s used only a few processors, in the 1990s, machines with thousands of processors began to appear and by the end of the 20th century, massively parallel supercomputers with tens of thousands of "off-the-shelf" processors were the norm [Science and Board (1989)] [Hill (2000)].

Systems with a massive number of processors generally take one of two paths: in one approach, e.g. in grid computing the processing power of a large number of computers in distributed, diverse administrative domains, is opportunistically used whenever a computer is available [Prodan (2007)]. In another approach, a large number of processors are used in close proximity to each other, e.g. in a supercomputer cluster. The use of multi-core processors combined with centralization is an emerging direction [Niu et al. (2005), Tan et al. (2011)]. Currently, Japan's K computer (a cluster) is the fastest in the world [The New York Times (2011)].

Supercomputers are used for highly computation-intensive tasks such as problems including quantum physics, weather forecasting, climate research, oil and gas exploration, molecular modelling (computing the structures and properties of chemical compounds, biological macromolecules, polymers, and crystals), and physical simulations (such as simulation of airplanes in wind tunnels, simulation of the detonation of nuclear weapons, and research into nuclear fusion).

High-throughput computing is a computer science term to describe the use of many computing resources over long periods of time to accomplish many computational tasks. The primary

objective is solving as many tasks as possible over a period of time.

There are many differences between high-throughput computing (HTC), high-performance computing (HPC), and many-task computing (MTC). HPC tasks are characterized as needing large amounts of computing power for short periods of time, whereas HTC tasks also require large amounts of computing, but for much longer times (months and years, rather than hours and days) [Beck (1997)]. HPC environments are often measured in terms of FLOPS. The HTC community, however, is not concerned about operations per second, but rather operations per month or per year. Therefore, the HTC field is more interested in how many jobs can be completed over a long period of time instead of how fast an individual job can complete.

As a general rule, HPC systems are tightly coupled parallel jobs, and as such they must execute within a particular site with low-latency interconnects. Conversely, HTC systems are independent, sequential jobs that can be individually scheduled on many different computing resources across multiple administrative boundaries. HTC systems achieve this using various grid computing technologies and techniques.

MTC aims to bridge the gap between HTC and HPC. MTC is reminiscent of HTC, but it differs in the emphasis of using many computing resources over short periods of time to accomplish many computational tasks (i.e. including both dependent and independent tasks), where the primary metrics are measured in seconds (e.g. FLOPS, tasks/s, MB/s I/O rates), as opposed to operations (e.g. jobs) per month. MTC denotes high-performance computations comprising multiple distinct activities, coupled via file system operations.

2.2 Large-scale distributed systems

A distributed system [Coulouris et al. (2005), Tanenbaum and van Steen (2007)] consists of multiple autonomous computers that communicate through a computer network. The computers interact with each other in order to achieve a common goal.

The word distributed in terms such as “distributed system”, “distributed programming”, and “distributed algorithm” originally referred to computer networks where individual computers were physically distributed within some geographical area [Lynch (1996)]. The terms are nowadays used in a much wider sense, even referring to autonomous processes that run on the same physical computer and interact with each other by message passing [Andrews (1999), Dolev (2000)].

While there is no single definition of a distributed system [Ghosh (2006)], the following defining properties are commonly used:

- There are several autonomous computational entities, each of which has its own local memory [Andrews (1999), Dolev (2000), Ghosh (2006), Lynch (1996), Peleg (2000)].
- The entities communicate with each other by message passing [Andrews (1999), Ghosh (2006), Peleg (2000)].

A distributed system may have a common goal, such as solving a large computational problem [Ghosh (2006), Peleg (2000)]. Alternatively, each computer may have its own user with individual needs, and the purpose of the distributed system is to coordinate the use of shared resources or provide communication services to the users [Ghosh (2006), Peleg (2000)].

Other typical properties of distributed systems include the following:

- The system has to tolerate failures in individual computers [Ghosh (2006), Lynch (1996), Peleg (2000)].

- The structure of the system (network topology, network latency, number of computers) is not known in advance, the system may consist of different kinds of computers and network links, and the system may change during the execution of a distributed program [Lynch (1996), Peleg (2000)].

- Each computer has only a limited, incomplete view of the system. Each computer may know only one part of the input [Ghosh (2006), Lynch (1996), Peleg (2000)].

2.2.1 World Wide Web

It is important to know that this is not a synonym for the Internet. The World Wide Web (WWW), or just "the Web", as ordinary people call it, is a subset of the Internet. The Web consists of pages that can be accessed using a Web browser. The Internet is the actual network of networks where all the information resides. Things like Telnet, FTP, Internet gaming, Internet Relay Chat (IRC), and e-mail are all part of the Internet, but are not part of the World Wide Web. The Hyper-Text Transfer Protocol (HTTP) is the method used to transfer Web pages to your computer. With hypertext, a word or phrase can contain a link to another Web site. All Web pages are written in the hyper-text markup language (HTML), which works in conjunction with HTTP [TechTerms.com (2013)].

The three architectural bases of the Web [W3C Technical Architecture Group (2004)] are:

- Identification. URIs are used to identify resources.

- Interaction. Web agents communicate using standardized protocols that enable interaction through the exchange of messages which adhere to a defined syntax and semantics. By entering a URI into a retrieval dialog or selecting a hypertext link, a client tells her browser to perform a retrieval action for the resource identified by the URI. For example, the browser sends an HTTP GET request (part of the HTTP protocol) to a server, via TCP/IP port 80, and the server sends back a message containing what it determines to be a representation of the resource as of the time that representation was generated. Note that this example is specific to hypertext browsing of information —other kinds of interaction are possible, both within browsers and through the use of other types of Web agent.

- Formats. Most protocols used for representation retrieval and/or submission make use of a sequence of one or more messages, which taken together contain a payload of representation data and metadata, to transfer the representation between agents. The choice of interaction protocol places limits on the formats of representation data and metadata that can be transmitted. HTTP, for example, typically transmits a single octet stream plus metadata, and uses the "Content-Type" and "Content-Encoding" header fields to further identify the format of the representation. In this scenario, the representation transferred is in XHTML, as identified by the "Content-type" HTTP header field containing the registered Internet media type name, "application/xhtml+xml". That Internet media type name indicates that the representation data can be processed according to the XHTML specification.

A client's browser is configured and programmed to interpret the receipt of an "application/x-html+xml" typed representation as an instruction to render the content of that representation according to the XHTML rendering model, including any subsidiary interactions (such as requests for external style sheets or in-line images) called for by the representation. In the scenario, the XHTML representation data received from the initial request instructs client's browser to also retrieve and render in-line the weather maps, each identified by a URI and thus causing an additional retrieval action, resulting in additional representations that are processed by the browser according to their own data formats (e.g., "application/svg+xml" indicates the SVG data format), and this process continues until all of the data formats have been rendered. The result of all of this processing, once the browser has reached an application steady-state that completes client's initial requested action, is commonly referred to as a "Web page".

2.2.2 Grid

The term "the Grid" [Foster et al. (2001, 2002a), Foster and Kesselman (2004)] was coined in the mid-1990s to denote a (then) proposed distributed computing infrastructure for advanced science and engineering. Much progress has since been made on the construction of such an infrastructure and on its extension and application to commercial computing problems. And while the term "Grid" has also been on occasion conflated to embrace everything from advanced networking and computing clusters to artificial intelligence, there has also emerged a good understanding of the problems that Grid technologies address, and at least a first set of applications for which they are suited.

Grid concepts and technologies were originally developed to enable resource sharing within scientific collaborations, first within early gigabit/sec testbeds [Catlett (1992), Smarr and Catlett (1992)] and then on increasingly larger scales [Beiriger et al. (2000), Brunett et al. (1998), Johnston et al. (1999), Stevens et al. (1997)]. Applications in this context include distributed computing for computationally demanding data analyzes (pooling of compute power and storage), the federation of diverse distributed datasets, collaborative visualization of large scientific datasets (pooling of expertise), and coupling of scientific instruments with remote computers and archives (increasing functionality as well as availability).

A common theme underlying these different usage modalities is a need for *coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations* [Foster et al. (2001)]. More recently, it has become clear that similar requirements arise in commercial settings, not only for scientific and technical computing applications but also for commercial distributed computing applications, including enterprise application integration and business-to-business partner collaboration over the Internet. Just as the Web began as a technology for scientific collaboration and was adopted for e-business, we see a similar trajectory for Grid technologies.

We thus argue that both science and industry can benefit from Grids. However, at the risk of stating the case too broadly, we make a more comprehensive statement. A primary purpose of information technology and infrastructure is to enable people to perform their daily tasks more efficiently or effectively. To the extent that these tasks are performed in collaboration with others, Grids are more than just a niche technology, but rather a direction in which our infrastructure must evolve if it is to support our social structures and the way work gets done in our society.

The success of the Grid to date owes much to the relatively early emergence of clean architectural principles, de facto standard software, aggressive early adopters with challenging application problems, and a vibrant international community of developers and users. This combination of factors led to a solid base of experience that has more recently driven the definition of the service-oriented Open Grid Services Architecture that today forms the basis for both open source and commercial Grid products.

2.2.2.1 Globus Toolkit

From 1997 onward, the open source Globus Toolkit™ version 2 (GT2) [Foster (2005a), Foster and Kesselman (1996), Foster (2005b)] emerged as the de facto standard for Grid computing [Foster and Kesselman (2004)]. Focusing on usability and interoperability, GT2 defined and implemented protocols, APIs, and services used in thousands of Grid deployments worldwide. By providing solutions to common problems such as authentication, resource discovery, and resource access, GT2 accelerated the construction of real Grid applications. Also by defining and implementing "standard" protocols and services, GT2 pioneered the creation of interoperable Grid systems and enabled significant progress on Grid programming tools. The GT2 protocol suite leveraged existing Internet standards for transport, resource discovery, and security. Some elements of the GT2 protocol suite were codified in formal technical specifications, reviewed within standards bodies, and instantiated in multiple implementations: notably, the GridFTP data transfer protocol [Allcock et al. (2003)] and elements of the Grid Security Infrastructure [Kesselman (2001), Tuecke et al. (2004)]. However, in general, GT2 "standards" were neither formal nor subject to public review. Similar comments apply to other important Grid technologies that emerged during this period, such as the Condor high-throughput computing system.

2.2.2.2 Open Grid Services Architecture (OGSA)

The year 2002 saw the emergence of the Open Grid Services Architecture (OGSA) [Foster et al. (2002b), Foster and Kesselman (2004)], a true community standard with multiple implementations, including, in particular, the OGSA-based GT 3.0, released in 2003. Building on and significantly extending GT2 concepts and technologies, OGSA firmly aligns Grid computing with broad industry initiatives in service-oriented architecture and Web services. In addition to defining a core set of standard interfaces and behaviors that address many of the technical challenges introduced previously, OGSA provides a framework within which one can define a wide range of interoperable, portable services. OGSA provides a foundation on which can be constructed a rich Grid technology ecosystem comprising multiple technology providers.

2.2.2.3 Managed, shared virtual systems

The definition of the initial OGSA technical specifications is an important step forward, but much more remains to be done before the full Grid vision is realized [Foster and Kesselman (2004)]. Building on OGSA's service-oriented infrastructure, we will see an expanding set of interoperable services and systems that address scaling to both larger numbers of entities and smaller device footprints, increasing degrees of virtualization, richer forms of sharing, and increased qualities of service via a variety of forms of active management. This work will draw increasingly heavily on

the results of advanced computer science research in such areas as peer-to-peer, knowledge-based [Berners-Lee et al. (2001)], and autonomic [Horn (2001)] systems.

We define a Grid as a system that coordinates distributed resources using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service. We examine the key elements of this definition:

- Coordinates distributed resources
- Using standard, open, general-purpose protocols and interfaces
- To deliver nontrivial qualities of service

2.2.2.4 gLite

gLite [EGEE Project (2013)] is a middleware computer software project for grid computing used by the CERN LHC experiments and other scientific domains. It was implemented by collaborative efforts of more than 80 people in 12 different academic and industrial research centers in Europe. gLite provides a framework for building applications tapping into distributed computing and storage resources across the Internet. The gLite services were adopted by more than 250 computing centres and used by more than 15000 researchers in Europe and around the world.

2.2.3 Volunteer computing and desktop Grids

Volunteer Computing (VC) and Desktop Grids (DG) [Choi et al. (2008)] is a paradigm in which large numbers of computers, volunteered by members of the general public, provide computing and storage resources [Anderson et al. (2005)]. Early volunteer computing projects include the Great Internet Mersenne Prime Search [Mersenne Research, Inc. (2013)], SETI@home [Anderson et al. (2002)], distributed.net [distributed.net (2013)] and Folding@home [Larson et al. (2004), Beberg et al. (2009)]. Volunteer computing is being used in high-energy physics, molecular biology, medicine, astrophysics, climate study, and other areas.

Since the late 1990's [INRIA/IN2P3 (2008)], Volunteer Computing systems, such as SETI@Home [Anderson et al. (2002)], have been the largest and most powerful distributed computing systems in the world, offering an abundance of computing power at a fraction of the cost of dedicated, custom-built supercomputers. Many applications from a wide range of scientific domains –including computational biology, climate prediction, particle physics, and astronomy– have utilized the computing power offered by Volunteer Computing and Desktop Grid systems. Volunteer Computing and Desktop Grid systems have allowed these applications to execute at a huge scale, often resulting in major scientific discoveries that would otherwise had not been possible.

The computing resources that power VC and DG are shared with the owners of the machines. Because the resources are volunteered, utmost care is taken to ensure that the VC and DG tasks do not obstruct the activities of each machine's owner; a VC or DG task is suspended or terminated whenever the machine is in use by another person. As a result, VC and DG resources are volatile in the sense that any number of factors can cause the task of a VC or DG application to not complete. These factors include mouse or keyboard activity, the execution of other user applications, machine reboots, or hardware failures. Moreover, VC and DG resources are heterogeneous in the sense that they differ in operating systems, CPU speeds, network bandwidth, memory and disk sizes. Consequently, the design of systems and applications that utilize these system is challenging.

2.2.3.1 BOINC

BOINC (Berkeley Open Infrastructure for Network Computing) [Anderson (2004)] is a middleware system for volunteer computing that makes it easy for scientists to create and operate public-resource computing projects. It supports diverse applications, including those with large storage or communication requirements. PC owners can participate in multiple BOINC projects, and can specify how their resources are allocated among these projects. BOINC is being used by a number of projects, including SETI@home, Climateprediction.net, LHC@home, Predictor@home, and Einstein@Home. Volunteers participate by running a BOINC client program on their computers. They can "attach" each computer to any set of projects, and can control the resource fraction devoted to each project.

Some projects require efficient data replication: Einstein@home [LIGO Scientific Collaboration (2009), LIGO Scientific Collaboration and Anderson (2009)] uses large (40 MB) input files, and a given input file may be sent to a large number of hosts (in contrast with projects like SETI@home [Werthimer et al. (2001), Paul (2002), Anderson et al. (2002)], where each input file is different).

The BOINC architecture [Anderson et al. (2005)] allows data servers to be located anywhere; they are simply web servers, and do not access the BOINC database. Current BOINC-based projects that use large files (Einstein@Home [American Physical Society (2013)] and Climateprediction.net [Oxford University (2013)]) use replicated and distributed data servers, located at partner institutions. The upload/download traffic is spread across the commodity Internet connections of those institutions.

BOINC-based projects are autonomous. Each project operates a server consisting of several components:

- Web interfaces for account and team management, message boards, and other features.
- A task server that creates tasks, dispatches them to clients, and processes returned tasks.
- A data server from which BOINC clients download input files and executables, and to which output files are uploaded.
- BOINC clients that download input files and executables, and upload output files.

These components share various data stored on disk, including relational databases and upload/download files (see Figure 2.1).

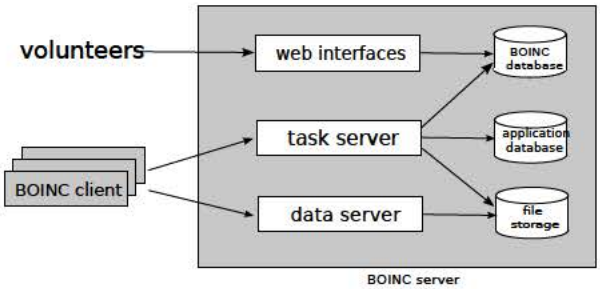


Figure 2.1: *A BOINC server consists of several components, sharing several forms of storage.*

Data servers handle file uploads using a certificate-based mechanism to ensure that only legitimate files, with prescribed size limits, can be uploaded. File downloads are handled by plain HTTP.

Files (associated with application versions, workunits, or results) have project-wide unique names and are immutable. Files can be replicated: the description of a file includes a list of URLs from which it may be downloaded or uploaded. Files can have associated attributes indicating, for example, that they should remain resident on a host after their initial use, that they must be validated with a digital signature, or that they must be compressed before network transfer.

When the BOINC client communicates with a scheduling server it reports completed work, and receives an XML document describing a collection of the above entities. The client then downloads and uploads files and runs applications; it maximizes concurrency, using multiple CPUs when possible and overlapping communication and computation. BOINC's computational system also provides a distributed storage facility (of computational inputs or results, or of data not related to distributed computation) as a by-product. This storage facility is much different from peer-to-peer storage systems such as Gnutella, PAST [Rowstron and Druschel (2001)] and Oceanstore [Kubiatowicz et al. (2000)]. In these systems, files can be created by any peer, and there is no central database of file locations. This leads to a set of technical problems (e.g. naming and file location) that are not present in the BOINC facility.

The BOINC architecture is based on a strict master/worker model (see Figure 2.2), with a central server responsible for dividing applications in thousands of small independent tasks and then distributing the tasks to the worker nodes as they request the work units. To simplify network communication and bypass any NAT problems that might arise with bidirectional communication, the centralized server never initiates communication with worker nodes: all communication is instantiated from the worker when more work is needed or results are ready for submission.

2.2.4 Cloud

The term "cloud computing" covers a range of delivery and service models [U.S. Department of Energy (2011)]. The common characteristic of these service models is an emphasis on pay-as-you-go and elasticity, the ability to quickly expand and collapse the utilized service as demand requires. Thus new approaches to distributed computing and data analysis have also emerged in conjunction with the growth of cloud computing. These include models like MapReduce and scalable key-value stores like Big Table [Chang et al. (2006, 2008)].

Cloud computing technologies and service models are attractive to scientific computing users due to the ability to get on-demand access to resources to replace or supplement existing systems, as well as the ability to control the software environment. Scientific computing users and resource providers servicing these users are considering the impact of these new models and technologies. In this section, we briefly describe the cloud service models and technologies to provide some foundation for the discussion.

2.2.4.1 Service models

Cloud offerings are typically categorized as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [U.S. Department of Energy (2011)]. Each of these models can play a role in scientific computing. The distinction between the service models is based on

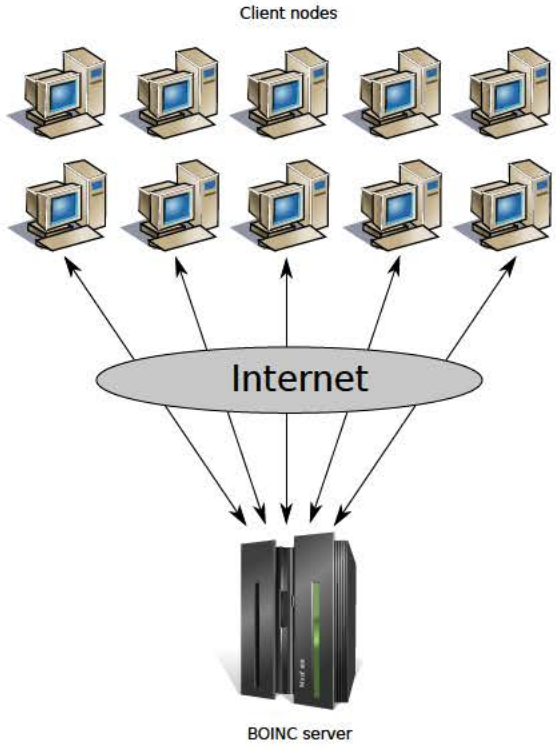


Figure 2.2: *Strict master/worker model in BOINC*

the layer at which the service is abstracted to the end user (e.g., hardware, system software, etc.). The end user then has complete control over the software stack above the abstracted level. Thus, in IaaS, a virtual machine or hardware is provided to the end user and the user then controls the operating system and the entire software stack. We describe each of these service models and visit existing examples in the commercial cloud space to understand their characteristics.

Infrastructure as a Service In the Infrastructure as a Service provisioning model, an organization outsources equipment including storage, hardware, servers, and networking components. The service provider owns the equipment and is responsible for housing, running, and maintaining it. In the commercial space, the client typically pays on a per-use basis for use of the equipment. Amazon Web Services is the most widely used IaaS cloud computing platform today. Amazon provides a number of different levels of computational power for different pricing. The primary methods for data storage in

Amazon EC2 are S3 and Elastic Block Storage (EBS). S3 is a highly scalable key-based storage system that transparently handles fault tolerance and data integrity. EBS provides a virtual storage device that can be associated with an elastic computing instance. S3 charges for space used per month, the volume of data transferred, and the number of metadata operations (in allotments of 1000). EBS charges for data stored per month. For both S3 and EBS, there is no charge for data transferred to and from EC2 within a domain (e.g., the U.S. or Europe).

Eucalyptus, OpenStack and Nimbus are open source software stacks that can be used to create a private cloud IaaS service. These software stacks provide an array of services that mimic many of the services provided by Amazon's EC2 including image management, persistent block storage,

virtual machine control, etc. The interface for these services is often compatible with Amazon EC2 allowing the same set of tools and methods to be used.

In Magellan, in conjunction with other synergistic activities, we use Amazon EC2 as the commercial cloud platform to understand and compare an existing cloud platform. We use Eucalyptus and OpenStack to set up a private cloud IaaS platform on Magellan hardware for detailed experimentation on providing cloud environments for scientific workloads. The IaaS model enables users to control their own software stack that is useful to scientists that might have complex software stacks.

Platform as a Service Platform as a Service (PaaS) provides a computing platform as a service, supporting the complete life cycle of building and delivering applications. PaaS often includes facilities for application design, development, deployment and testing, and interfaces to manage security, scalability, storage, state, etc. Windows Azure, Hadoop, and Google App Engine are popular PaaS offerings in the commercial space.

Windows Azure is Microsoft's offering of a cloud services operating system. Azure provides a development, service hosting, and service management environment. Windows Azure provides on-demand compute and storage resources for hosting applications to scale costs. The Windows Azure platform supports two primary virtual machine instance types—the Web role instances and the Worker role instances. It also provides Blobs as a simple way to store data and access it from a virtual machine instance. Queues provide a way for Worker role instances to access the work quantum from the Web role instance. While the Azure platform is primarily designed for web applications, its use for scientific applications is being explored [Li et al. (2010), Qiu et al. (2009), Ekanayake et al. (2010)].

Hadoop is another well-known example of PaaS, that will be described later in section 2.3.5.1. Hadoop provides a platform for managing loosely coupled data-intensive applications.

PaaS provides users with the building blocks and semantics for handling scalability, fault tolerance, etc. in their applications.

Software as a Service Software as a Service provides access to an end user for an application or software that has a specific function. Examples in the commercial space include services like SalesForce and Gmail. In our project activities, we use the Windows Azure BLAST service to run BLAST jobs on the Windows Azure platform. Science portals can also be viewed as providing a Software as a Service, since they typically allow remote users to perform analysis or browse data sets through a web interface. This model can be attractive since it allows the user to transfer the responsibility of installing, configuring, and maintaining an application and shields the end-user from the complexity of the underlying software.

Hardware as a Service Hardware as a Service (HaaS) is also known as "bare-metal provisioning". The main distinction between this model and IaaS is that the user-provided operating system software stack is provisioned onto the raw hardware, allowing the users to provide their own custom hypervisor, or to avoid virtualization completely, along with the performance impact of virtualization of high-performance hardware such as InfiniBand. The other difference between HaaS and the other service models is that the user "leases" the entire resource; it is not shared with other users within a virtual space. With HaaS, the service provider owns the equipment and

is responsible for housing, running, and maintaining it. HaaS provides many of the advantages of IaaS and enables greater levels of control on the hardware configuration.

2.2.4.2 Deployment models

According to the NIST definition, clouds can have one of the following deployment models, depending on how the cloud infrastructure is operated: (a) public, (b) private, (c) community, or (d) hybrid.

Public Cloud. Public clouds refer to infrastructure provided to the general public by a large industry selling cloud services. Amazon's cloud offering would fall in this category. These services are on a pay-as-you-go basis and can usually be purchased using a credit card.

Private Cloud. A private cloud infrastructure is operated solely for a particular organization and has specific features that support a specific group of policies. Cloud software stacks such as Eucalyptus, OpenStack, and Nimbus are used to provide virtual machines to the user. In this context, Magellan can be considered a private cloud that provides its services to DOE Office of Science users.

Community Cloud. A community cloud infrastructure is shared by several organizations and serves the needs of a special community that has common goals. FutureGrid [Indiana University (2013)] can be considered a community cloud.

Hybrid Cloud. Hybrid clouds refer to two or more cloud infrastructures that operate independently but are bound together by technology compliance to enable application portability.

2.2.5 Internet of Things

According to [Atzori et al. (2010)], the Internet of Things (IoT) is a novel paradigm that is rapidly gaining ground in the scenario of modern wireless telecommunications. The basic idea of this concept is the pervasive presence around us of a variety of things or objects – such as Radio-Frequency IDentification (RFID) tags, sensors, actuators, mobile phones, etc. – which, through unique addressing schemes, are able to interact with each other and cooperate with their neighbors to reach common goals [Giusto et al. (2010)].

Unquestionably, the main strength of the IoT idea is the high impact it will have on several aspects of everyday-life and behavior of potential users. From the point of view of a private user, the most obvious effects of the IoT introduction will be visible in both working and domestic fields. In this context, domotics, assisted living, e-health, enhanced learning are only a few examples of possible application scenarios in which the new paradigm will play a leading role in the near future. Similarly, from the perspective of business users, the most apparent consequences will be equally visible in fields such as, automation and industrial manufacturing, logistics, business/process management, intelligent transportation of people and goods.

"Internet of Things" semantically means "a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols" [Bassi et al. (2008)]. This

implies a huge number of (heterogeneous) objects involved in the process.

The very first definition of IoT derives from a "Things oriented" perspective; the considered things were very simple items: Radio-Frequency IDentification (RFID) tags. The terms "Internet of Things" is, in fact, attributed to The Auto-ID Labs [Sarma and Fleisch (2015)], a world-wide network of academic research laboratories in the field of networked RFID and emerging sensing technologies.

2.2.6 Wireless Sensor Networks

A wireless sensor network (WSN) consists of spatially distributed autonomous sensors to monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants and to cooperatively pass their data through the network to a main location. The more modern networks are bi-directional, also enabling control of sensor activity. The development of wireless sensor networks was motivated by military applications such as battlefield surveillance; today such networks are used in many industrial and consumer applications, such as industrial process monitoring and control, machine health monitoring, and so on.

The WSN is built of "nodes" —from a few to several hundreds or even thousands, where each node is connected to one (or sometimes several) sensors. Each such sensor network node has typically several parts: a radio transceiver with an internal antenna or connection to an external antenna, a microcontroller, an electronic circuit for interfacing with the sensors and an energy source, usually a battery or an embedded form of energy harvesting. A sensor node might vary in size from that of a shoebox down to the size of a grain of dust, although functioning "motes" of genuine microscopic dimensions have yet to be created. The cost of sensor nodes is similarly variable, ranging from a few to hundreds of dollars, depending on the complexity of the individual sensor nodes. Size and cost constraints on sensor nodes result in corresponding constraints on resources such as energy, memory, computational speed and communications bandwidth. The topology of the WSNs can vary from a simple star network to an advanced multi-hop wireless mesh network. The propagation technique between the hops of the network can be routing or flooding [Dargie (2010), Sohraby (2007)].

2.2.7 Peer-to-peer

Peer-to-peer (P2P) computing or networking is a distributed application architecture that partitions tasks or workloads among peers. Peers are equally privileged, equipotent participants in the application. They are said to form a peer-to-peer network of nodes.

Peers make a portion of their resources, such as processing power, disk storage or network bandwidth, directly available to other network participants, without the need for central coordination by servers or stable hosts [Schollmeier (2001)]. Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model where only servers supply (send), and clients consume (receive).

The peer-to-peer application structure was popularized by file sharing systems like Napster. The concept has inspired new structures and philosophies in many areas of human interaction. Peer-to-peer networking is not restricted to technology, but covers also social processes with a peer-to-peer dynamic. In such context, social peer-to-peer processes are currently emerging throughout society.

2.3 Data-intensive computing paradigms

Three related distributed data-intensive research areas that share similar requirements, functions, and characteristics are described in the following sections [Venugopal et al. (2006)].

2.3.1 Data Grids

Scientific applications in domains as diverse as high energy physics, molecular modelling, and earth sciences involve the production of large datasets from simulations or from large-scale experiments [Venugopal et al. (2005, 2006)]. Collectively, these large scale applications have come to be known as part of e-Science [Hey and Trefethen (2002)], a discipline that envisages using high-end computing, storage, networking, and Web technologies together to facilitate collaborative, data-intensive scientific research.

Data Grids [Chervenak et al. (1999), Hoschek et al. (2000)] primarily deal with providing services and infrastructure for distributed data-intensive applications that need to access, transfer, and modify massive datasets stored in distributed storage resources.

According to Venugopal et al. [Venugopal et al. (2006)], a few studies have investigated and surveyed Grid research in the recent past. Krauter et al. [Krauter et al. (2002)] present a taxonomy of various Grid resource management systems that focuses on the general resource management architectures and scheduling policies. Specifically for Data Grids, Bunn and Newman [Bunn and Newman (2003)] provide an extensive survey of projects in High Energy Physics, while Qin and Jiang [Qin and Jiang (2003)] produce a compilation that concentrates more on the constituent technologies. Moore and Merzky [Moore and Merzky (2003)] identify functional requirements (features and capabilities) and components of a persistent archival system. In contrast to these articles, Finkelstein et al. [Finkelstein et al. (2004)] spell out requirements for Data Grids from a software engineering perspective and elaborate on the impact that these have on architectural choices. A similar characterisation has been performed by Mattmann et al. [Mattmann et al. (2005)].

A Data Grid provides services that help users discover, transfer, and manipulate large datasets stored in distributed repositories and also, create and manage copies of these datasets. At the minimum, a Data Grid provides two basic functionalities: a high-performance, reliable data transfer mechanism, and a scalable replica discovery and management mechanism [Chervenak et al. (1999)].

2.3.2 Content Delivery Networks

A Content Delivery Network (CDN) [Davison (2001), Dilley et al. (2002)] consists of a "collection of (nonorigin) servers that attempt to offload work from origin servers by delivering content on their behalf" [Krishnamurthy et al. (2001)]. That is, within a CDN, client requests are satisfied from other servers distributed around the Internet (also called edge servers) that cache the content originally stored at the source (origin) server. A client request is rerouted from the main server to an available server closest to the client likely to host the content required [Dilley et al. (2002)].

Content Delivery Networks [Pathan and Buyya (2007)] provide services that improve network performance by maximizing bandwidth, improving accessibility and maintaining correctness through content replication. They offer fast and reliable applications and services by distributing content to cache or edge servers located close to users. A CDN has some combination of

content-delivery, request-routing, distribution and accounting infrastructure. The content-delivery infrastructure consists of a set of edge servers (also called surrogates) that deliver copies of content to end-users. The request-routing infrastructure is responsible to directing client request to appropriate edge servers. It also interacts with the distribution infrastructure to keep an up-to-date view of the content stored in the CDN caches. The distribution infrastructure moves content from the origin server to the CDN edge servers and ensures consistency of content in the caches. The accounting infrastructure maintains logs of client accesses and records the usage of the CDN servers. This information is used for traffic reporting and usage-based billing. In practice, CDNs typically host static content including images, video, media clips, advertisements, and other embedded objects for dynamic Web content. Typical customers of a CDN are media and Internet advertisement companies, data centers, Internet Service Providers (ISPs), online music retailers, mobile operators, consumer electronics manufacturers, and other carrier companies. Each of these customers wants to publish and deliver their content to the end-users on the Internet in a reliable and timely manner. A CDN focuses on building its network infrastructure to provide the following services and functionalities: storage and management of content; distribution of content among surrogates; cache management; delivery of static, dynamic and streaming content; backup and disaster recovery solutions; and monitoring, performance measurement and reporting.

2.3.3 Peer-to-Peer networks

Peer-to-peer (P2P) networks [Oram (2001)] are formed by ad hoc aggregation of resources to form a decentralized system within which each peer is autonomous and depends on other peers for resources, information, and forwarding requests. The primary aims of a P2P network are to ensure scalability and reliability by removing the centralized authority, to ensure redundancy, to share resources, and to ensure anonymity. An entity in a P2P network can join or leave anytime and, therefore, algorithms and strategies have to be designed keeping in mind the volatility and requirements for scalability and reliability. P2P networks have been designed and implemented for many target areas such as compute resource sharing (e.g., SETI@Home [Werthimer et al. (2001), Anderson et al. (2002)], Compute Power Market [Buyya and Vazhkudai (2001)]), content and file sharing (Napster, Gnutella, Kazaa [Subramanian and Goodman (2005)]), and collaborative applications such as instant messengers (Jabber [Jabber.org (2013)]). Milojicic et al. [Milojicic et al. (2003)] present a detailed taxonomy and survey of peer-to-peer systems. And later, Androutsellis-Theotokis and Spinellis [Androutsellis-Theotokis and Spinellis (2004)] present a survey of peer-to-peer content distribution technologies.

2.3.4 Distributed databases

A distributed database (DDB) [Ceri and Pelagatti (1984), Özsu and Valduriez (1999)] is a logically organized collection of data stored at different sites of a computer network. Each site has a degree of autonomy, is capable of executing a local application, and also participates in the execution of a global application. A distributed database can be formed either by taking an existing single site database and splitting it over different sites (top-down approach) or by federating existing database management systems so that they can be accessed through a uniform interface (bottom-up approach) [Sheth and Larson (1990)]. The latter are also called multidatabase systems.

Recently, Google developed Spanner. Spanner [Corbett et al. (2012)] is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system

to distribute data at global scale and support externally-consistent distributed transactions.

2.3.5 Big Data

Big data [Lynch (2008), Howe et al. (2008), Manyika et al. (2011)] is an open concept that has multiple definitions. According to Kusnetzky, "In simplest terms, the phrase refers to the tools, processes and procedures allowing an organization to create, manipulate, and manage very large data sets and storage facilities" [Kusnetzky (2010)]. But "big" does not refer exactly to big volume only, but also to big complexity [MIKE2.0 (2013)]. And that combination of volume and complexity is what makes the data set difficult to process using traditional data processing tools.

The most well-known programming model for Big Data is probably MapReduce, that is now discussed.

2.3.5.1 MapReduce

MapReduce [Dean and Ghemawat (2008, 2010)] is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system [Dean and Ghemawat (2004)]. MapReduce is tightly coupled with Google File System.

The Google File System [Ghemawat et al. (2003a,b)] is a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients. It is widely deployed within Google as the storage platform for the generation and processing of data used by Google's services as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

A popular open-source implementation of MapReduce is Hadoop [The Apache Software Foundation (2015b)]. Hadoop [White (2011)] is an open-source software that provides capabilities to harness commodity clusters for distributed processing of large data sets through the MapReduce [Dean and Ghemawat (2008)] model. The Hadoop streaming model allows one to create map-and-reduce jobs with any executable or script as the mapper and/or the reducer. This is the most suitable model for scientific applications that have years of code in place capturing complex scientific processes.

The Hadoop Distributed File System (HDFS) is the primary storage model used in Hadoop. HDFS is modeled after the Google File system [Ghemawat et al. (2003a,b)] and has several features that are specifically suited to Hadoop/MapReduce. Those features include exposing data locality and data replication. Data locality is a key mechanism that enables Hadoop to achieve good scaling and performance, since Hadoop attempts to locate computation close to the data. This is

particularly true in the map phase, which is often the most I/O intensive phase.

2.4 Data transport technologies in large-scale distributed systems

In this section, various projects involved in data transport over distributed systems are discussed. The data transport technologies studied here range from protocols such as FTP to over-lay methods, such as Internet Backplane Protocol, to file I/O mechanisms [Venugopal et al. (2006)].

2.4.1 HTTP

The Hypertext Transfer Protocol (HTTP) [Fielding et al. (1999)] is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred. HTTP has been in use by the World-Wide Web global information initiative since 1990.

The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible entity-body content.

2.4.1.1 REST

Representational State Transfer (REST) [Fielding (2000), Fielding and Taylor (2002)] is a coordinated set of architectural constraints that attempts to minimize latency and network communication, while at the same time maximizing the independence and scalability of component implementations. This is achieved by placing constraints on connector semantics, where other styles have focused on component semantics. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries, in order to meet the needs of an Internet-scale distributed hypermedia system.

The name "Representational State Transfer" is intended to evoke an image of how a well-designed Web application behaves: a network of Web pages forms a virtual state machine, allowing a user to progress through the application by selecting a link or submitting a short data-entry form, with each action resulting in a transition to the next state of the application by transferring a representation of that state to the user [Fielding and Taylor (2002)].

The modern Web is one instance of a REST-style architecture. Although Web-based applications can include access to other styles of interaction, the central focus of its protocol and performance concerns is distributed hypermedia. REST elaborates only those portions of the architecture that are considered essential for Internet-scale distributed hypermedia interaction. Areas for improvement of the Web architecture can be seen where existing protocols fail to express all of the potential semantics for component interaction, and where the details of syntax can

be replaced with more efficient forms without changing the architecture capabilities [Fielding and Taylor (2002)].

2.4.2 GASS

Global Access to Secondary Storage (GASS) [Bester et al. (1999)] is a data access mechanism provided within the Globus toolkit for reading local data at remote machines and for writing data to remote storage and moving it to a local disk. The goal of GASS is to provide a uniform remote I/O interface to applications running at remote resources, while keeping the functionality demands on both the resources and the applications limited.

2.4.3 IBP

Internet Backplane Protocol (IBP) [Plank et al. (1999), Bassi et al. (2002)] allows applications to optimize data transfer and storage operations by controlling data transfer explicitly by storing the data at intermediate locations. IBP uses a store-and-forward protocol to move data around the network. Each of the IBP nodes has a temporary buffer into which data can be stored for a fixed amount of time. Applications can manipulate these buffers so that data is moved to locations close to where it is required.

2.4.4 FTP

The File Transfer Protocol (FTP) [Postel and Reynolds (1985)], defined by RFC 959, is one of the fundamental protocols for data movement in the Internet. FTP is, therefore, ubiquitous, and every operating system ships with an FTP client. FTP separates the process of data transfer into two channels, the control channel used for sending commands and replies between a client and a server, and the data channel through which the actual transfer takes place.

2.4.5 GridFTP

GridFTP [Allcock et al. (2001a,b, 2002, 2003), Mandrichenko et al. (2005)] is a data transfer protocol defined by Global Grid Forum Recommendation GFD.020 [Allcock et al. (2003)], RFC 959, RFC 2228, RFC 2389, and a draft before the IETF FTP working group. GridFTP extends the default FTP protocol by providing features that are required in a Data Grid environment. The aim of GridFTP is to provide secure, robust, fast, and efficient transfer of (especially bulk) data in Grid environments. The Globus Toolkit™ provides the most commonly used implementation of this protocol, though others do exist (primarily tied to proprietary internal systems). GridFTP extends the FTP protocol, and includes the following features:

- Grid Security Infrastructure (GSI) support, and Kerberos-based authentication
- Third-party control and data transfer
- Parallel data transfer using multiple TCP streams
- Striped data transfer using multiple servers
- Partial file transfer and support for reliable and restartable data transfer

2.4.6 Globus XIO

The Globus™ eXtensible Input/Output (XIO) System [Allcock et al. (2005), Kettimuthu et al. (2008)] is a simple Open/Close/Read/Write (OCRW) abstraction layer to transport protocols that was designed to ease the development of communication protocols to optimize use of and access to the underlying fabric of the Grid. To application developers, Globus XIO is an API that enables different I/O problems to be presented uniformly as a simple OCRW interface with a single set of semantics. To network protocol developers, it is a support framework for developing communication protocols. To mass storage vendors, it is an interface that enables an existing application written with XIO to access their hardware.

Globus XIO [Kettimuthu et al. (2008)] comprises two main components: framework and drivers. Figure 2.3 illustrates the architecture.

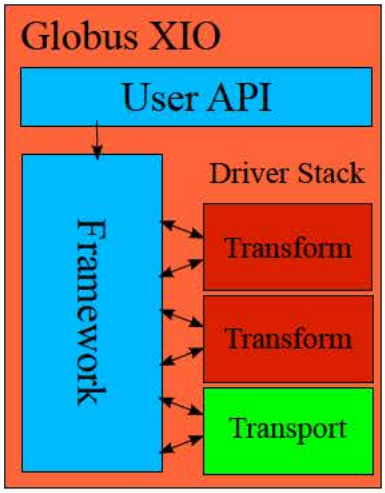


Figure 2.3: *Globus XIO Architecture*

The Globus XIO framework manages I/O operation requests that an application makes via the user API. The framework does not manipulate or deliver the data in an I/O operation; the drivers do all of that work. The framework's job is to manage requests and map them to the drivers' interface.

A driver in Globus XIO is responsible for manipulating and transporting the user's data. There are two types of drivers: transform and transport. Transform drivers are those that manipulate the data buffers passed to it via the user API and the XIO framework. Transport drivers are those that are capable of sending the data over a wire.

Drivers can be grouped into a stack. When an I/O operation is requested, the Globus XIO framework passes the operation request to every driver in the order the drivers are in the stack. When the bottom-level driver (the transport driver) finishes shipping the data, it passes the request completion notification back to the XIO framework. Globus XIO then delivers the notification back up the stack in this manner until it reaches the top, at which point the application is notified that its request is completed.

In a driver stack there must only be one transport driver, and it must be at the bottom of the stack. The reason is that the transport driver is what actually moves the bits on a wire. Any number of transform drivers can be in a stack. Good examples of transform drivers are security

wrappers and compression.

Transport drivers are expected to implement some kind of common semantic behavior. Transport drivers that follow the same semantics (for e.g., TCP and UDP) can be interchanged without user code change. The GridFTP driver was designed to follow a 'file-like' semantic, so it could be swapped out with the file driver without user code change.

2.5 Data replication

Providing efficient data access and maximum data availability is a challenging task. To achieve this task, data is replicated to different sites [Amjad et al. (2012)].

2.5.1 Benefits of data replication strategies

The benefits of data replication strategies can be summarized in:

Availability All the replication strategies aim to provide maximum availability. Rather, it would be better to say that replication is the only way to improve availability of data: generally in all distributed database environments and specifically in data grids.

Reliability When replication increases the availability, the reliability is improved as well. The more the number of replicas more is the chance that user's request will be serviced properly, and hence systems is more reliable.

Scalability It is another important metric that must be considered by a replication algorithm. The extent to which scalability can be provided depends upon the architecture chosen for the data grid. Different architectural models support different levels of scalability. That means, scalability is more dependent on model than replication algorithm.

Adaptability This is a very important parameter which must be provided by a replication strategy. The nature of the grid is very dynamic. Nodes keep on entering and leaving the grid very frequently. The replication algorithm must be adaptive to provide support to all nodes present in a data grid at any given time.

Performance As the availability of data increases the performance of the data grid environment increases.

2.5.2 Static replication strategies

In a static replication strategy [Amjad et al. (2012)], the number of replicas and the host node is chosen statically at the start of the life cycle, no more replicas are created or migrated after that [Tatebe et al. (2002), Chervenak et al. (2002)].

However, static replication strategies are worse that dynamic strategies because the latter can make intelligent decisions about the placement of data depending upon the information of the environment. Hence, the presence in the literature of works about static replication is smaller [Loukopoulos and Ahmad (2000), Khan and Ahmad (2008)].

2.5.3 Dynamic replication strategies

Dynamic replication strategies [Amjad et al. (2012)] adapt to changes in user request pattern, storage capacity and bandwidth and can create replicas on new nodes and can delete replicas that are no longer required depending upon the global information of the data grid [Ranganathan and Foster (2001), Yuan et al. (2007), Lamehamedi et al. (2003), Lee and Weissman (2001)].

Dynamic replication [Amjad et al. (2012)] is an optimization technique which aims to reduce the average job execution time. It ensures high availability of data, and improved usage of network bandwidth available. There are certain issues which a data replication technique must address during replication according to the constraints of a specific situation:

Dynamic nature The nature of the grid is very dynamic and users can join and leave a grid at any time. So the number of participants present in a grid at any given time can vary. The data replication algorithm must be adaptive to the changing size of the grid in order to provide better results.

Grid architecture The replication technique is highly dependent upon architecture of the grid. A data grid can be supported by many different architectures. It can be a multi-tier architecture; a tree like structure in which the nodes are arranged in a tree like hierarchy. For example, the data grid of the GriPhyN project in which tier 0 is the main data source (CERN), tier 1 contains the national centers, tier 2 the regional centers, tier 3 the work groups and finally at tier 4 are the desktops. Alternatively it can be a graph like topology, in which any node can be connected to any other node without any restrictions of tree topology. It can be a peer to peer topology, or it can be any hybrid model. A replication technique is designed according to the architecture in question.

Decision making Data replication involves a very critical decision, i.e. when to replicate data, which files should be replicated, and where the replica should be placed. Depending on the answers, different replication strategies have been evolved.

Available storage space Although the storage devices have now become very cheap, the replication strategies must still keep into account the amount of available storage space before creating a replica. In case the available storage is not sufficient enough to store a replica, a replacement strategy is adopted.

Cost of replication The replication strategy must ensure that the benefit of the replication is higher than the cost of replication.

2.5.4 RLS

Replica Location Service (RLS) [Chervenak et al. (2002)] is a system that maintains information about physical locations of copies of data. The main components of RLS are the Local Replica Catalog (LRC) which maps the logical representation to the physical locations and the Replica Location Index (RLI) which indexes the catalog itself. The actual data is represented by a logical file name (LFN) and contain some information such as the size of the file, its creation date, and any other such metadata that might help users to identify the files that they seek. A logical file has a mapping to the actual physical location(s) of the data file and its replicas, if any. The physical

location is identified by a unique physical file name (PFN) which is a URL (Uniform Resource Locator) to the data file on storage. Therefore, a LRC provides the PFN corresponding to an LFN.

2.6 Distributed storage

Storage plays a fundamental role in computing, a key element, ever present from registers and RAM to hard-drives and optical drives. Functionally, storage may service a range of requirements, from caching (expensive, volatile and fast) to archival (inexpensive, persis- tent and slow). Combining networking and storage has created a platform with numerous possibilities allowing Distributed Storage Systems (DSS) to adopt roles vast and varied which fall well beyond data storage [Placek and Buyya (2006)].

There are many distributed file systems: AFS [Howard et al. (1988)], NFS [Sandberg et al. (1985)], Coda [Satyanarayanan et al. (1990), Kistler and Satyanarayanan (1992)], xFS [Anderson et al. (1995, 1996)], GFS [Soltis et al. (1996, 2002)], etc. But there are not many mechanisms suitable for large-scale distributed systems like Grid environments, volunteer computing or desktop grids. In this section, several distributed storage and file systems used within distributed systems are studied.

2.6.1 Grid DataFarm (Gfarm)

The Grid Datafarm (Gfarm) [Tatebe et al. (2002, 2005, 2010)] is an architecture for petascale data-intensive computing on the Grid. This model specifically targets applications where data primarily consists of a set of records or objects which are analyzed independently. Gfarm takes advantage of this access locality to achieve a scalable I/O bandwidth using an enhanced parallel filesystem integrated with process scheduling and file distribution. It provides a global, Grid-enabled, fault-tolerant parallel filesystem whose I/O bandwidth scales to the TB/s range, and which incorporates fast file transfer techniques and wide-area replica management.

Large-scale data-intensive computing frequently involves a high degree of data access locality. To exploit this access locality, Gfarm schedules programs on nodes where the corresponding segments of data are stored to utilize local I/O scalability, rather than transferring the large-scale data to compute nodes. Gfarm consists of the Gfarm filesystem, the Gfarm process scheduler, and Gfarm parallel I/O APIs. Together, these components provide a Grid-enabled solution to a class of data-intensive problems.

A Gfarm file is a large-scale file that is divided into fragments and distributed across the disks of the Gfarm filesystem, and which will be accessed in parallel. The Gfarm filesystem is an extension of a striping parallel system in that each file fragment has an arbitrary length and can be stored on any node.

Every Gfarm file is basically write-once. Applications are assumed to create a new file instead of updating an existing file. The Gfarm parallel I/O API supports read/write open, which is internally implemented by versioning and creating a new file. This is because 1) large-scale data is seldom updated (most data is write-once and read-many), and 2) data can be recovered by replication, or by recomputation using a command history log.

2.6.2 Grid File Access Library (GFAL)

The Grid File Access Library (GFAL) [CERN (2015)] is a library that offers to the user a POSIX-like interface to access data on various flavours of Storage Elements offering an SRM interface. GFAL is interfaced to SRM-compliant back-ends (both v1.1 and v2.2) and storage systems such as Castor, dCache or DPM in which case the relevant protocol is used transparently behind the scenes. Using information published in the information system, it resolves relevant abstract domain data/file names so that the physical data access as well as the end-points of services are achievable transparently. It allows and unifies access to various types of items such as: LFN, GUID, SURL, SRM and TURL or local path. In addition, some of the crucial, yet common, backend calls are exposed through the library so that users are not limited to POSIX mapping to do specific calls e.g. to reserve space or pin a file. The pluggable architecture of the library permits dynamic change of the versions of some of the supported protocols (i.e. rfio, dCache) without need of redeployment [European Middleware Initiative (2015, 2013b,a)].

2.6.3 Legion I/O

Legion [Chapin et al. (1999)] is a object-oriented grid middleware for providing a single system image across a collection of distributed resources. The I/O mechanism within Legion [White et al. (2000)] aims to provide transparent access to files stored on distributed resources through APIs and daemons that can be used by native and legacy applications alike.

2.6.4 SRB I/O

The Storage Resource Broker (SRB) [Baru et al. (1998)] developed at the San Diego Supercomputing Center (SDSC) focuses on providing a uniform and transparent interface to heterogenous storage systems that include disks, tape archives, and databases.Data transport within SRB provides features such as parallel data transfers for performing bulk data transfer operations across geographically distributed sites.

The purpose of the SRB is to enable the creation of shared collections through management of consistent state information, latency management, load levelling, logical resources usage, and multiple access interfaces [Baru et al. (1998), Rajasekar et al. (2003)]. SRB also aims to provide a unified view of the data files stored in disparate media and locations by providing the capability to organize them into virtual collections independent of their physical location and organization. It provides a large number of capabilities that are not only applicable to Data Grids but also for collection building, digital libraries, and persistent archival applications.

2.6.5 Bigtable

Bigtable [Chang et al. (2006, 2008)] is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products.

2.6.6 Ceph

Ceph [Weil et al. (2006a)] is a distributed storage system designed for scalability, reliability, and performance. The system is based on a distributed object storage service called RADOS (reliable autonomic distributed object store) that manages the distribution, replication, and migration of objects. On top of that reliable storage abstraction Ceph builds a range of services, including a block storage abstraction (RBD, or Rados Block Device) and a cache-coherent distributed file system (CephFS).

Data objects are distributed across Object Storage Devices (OSD), which refers to either physical or logical storage units, using CRUSH [Weil et al. (2006b)], a deterministic hashing function that allows administrators to define flexible placement policies over a hierarchical cluster structure (e.g., disks, hosts, racks, rows, datacenters). The location of objects can be calculated based on the object identifier and cluster layout (similar to consistent hashing [Karger et al. (1997)]), thus there is no need for a metadata index or server for the RADOS object store. A small cluster of monitors (ceph-mon daemons) use Paxos to provide consensus on the current cluster layout, but do not need to explicitly coordinate migration or recovery activities.

CephFS builds a distributed cache-coherent file system on top of the object storage service provided by RADOS. Files are striped across replicated storage objects, while a separate cluster of metadata servers (ceph-mds daemons) manage the file system namespace and coordinate client access to files.

Ceph metadata servers store all metadata in RADOS objects, which provides a shared, highly-available, and reliable storage backend. Unlike many other distributed file system architectures, Ceph also embeds inodes inside directories in the common case, allowing entire directories to read from RADOS into the metadata server cache or prefetched into the client cache using a single request.

Client hosts that mount the file system communicate with metadata servers to traverse the namespace and perform file I/O by reading and writing directly to RADOS objects that contain the file data. The metadata server cluster periodically adjusts the distribution of the namespace across the MDS cluster by migrating responsibility for arbitrary subtrees of the hierarchy between a dynamic pool of active ceph-mds daemons. This dynamic subtree partitioning [Weil et al. (2004)] strategy is both adaptive and highly scalable, allowing additional metadata server daemons to be added or removed at any time, making it ideally suited both for large-scale workloads with bursty workloads or general purpose clusters whose workloads grow or contract over time [Wang et al. (2013)].

However, in Ceph, the metadata are stored in memory cache in MDS (metadata server). As for safety, MDS must commit journal to the OSD. The synchronous I/O fails to achieve the high performance when multiple clients upload or download thousands of files simultaneously [Duan et al. (2015)].

2.6.7 GlusterFS

GlusterFS [Red Hat, Inc (2015)] is a clustered file-system for scaling the storage capacity of many servers to several peta-bytes. It aggregates various storage servers or bricks over an interconnect such as InfiniBand or TCP/IP into one large parallel network file system. GlusterFS in its default configuration does not stripe the data, but instead distributes the namespace across all the servers.

Internally, GlusterFS is based on the concept of translators. Translators may be applied at either the client or the server. Translators exist for Read Ahead and Write Behind. In terms of design, a small portion of GlusterFS is in the kernel and the remaining portion is in userspace. The calls are translated from the kernel VFS to the userspace daemon through the Filesystem in UserSpace (FUSE) [Noronha and Panda (2008)].

[Donvito et al. (2014)] points out that GlusterFS is a storage technology that permits, starting from several volumes hosted on different servers, the construction of a distributed replicated network file-system, fully POSIX compliant, also with support of new storage paradigms such as Block Storage and Object Storage. GlusterFS stores the data on stable kernel file-systems like ext4, xfs, etc.; it does not use an additional metadata server for the files metadata, using instead a unique hash tag for each file, stored within the file-system itself.

In the Gluster terminology a volume is the share that the servers, that host the actual kernel space file-system in which the data will be stored, expose to the clients. Each volume can be built by several subvolumes, generally hosted by different servers. A subvolume is built by a brick, the storage file-system that has been assigned to the volume, processed by at least one translator. A translator connects to one or more subvolumes, does something with them, and offers a subvolume connection.

With these basic concepts one can build 3 types of complex volumes: distributed, replicated and striped. The most basic volume is a distribute only volume, that simply spread the data across the available bricks, so that over 100 files written on a volume built by two bricks, an average fifty will end up on one brick, and fifty on the other. If the bricks are hosted on two different servers, we have something similar to RAID0 for physical disks, with all the pros (increased velocity) and cons (increased fragility of the volume). With the replicated volume GlusterFS can transparently replicate the data with the multiplicity chosen at the volume creation, when it is possible to set the number of file replicas that the volume must contain. Obviously this setup is particularly useful if the bricks are located on different servers.

It is also possible to mix the basic volume types, so for example one can build a distributed-replicated volume, that distributes the data across multiple servers and replicates them in order to obtain an increased availability [Donvito et al. (2014)].

However GlusterFS still requires to install some components in all nodes [Beloglazov et al. (2012)].

2.6.8 Hadoop Distributed File System (HDFS)

As stated by [Donvito et al. (2014)], Apache Hadoop is an open-source software framework developed in Java that allows distributed processing of large data sets across clusters of computers using simple programming models. It is composed of several modules such as Hadoop Yarn and Hadoop MapReduce for cluster resource management and parallel processing, Hadoop Distributed File System (HDFS), inspired by the Google File System [Ghemawat et al. (2003a,b)], that provides high-throughput access to application data and other related sub-projects such as Cassandra, HBase, Zookeeper, etc.

HDFS [The Apache Software Foundation (2015a)] has a master/slave architecture. The NameNode is the master server that manages file-system namespace and regulates access to files by clients. It can be replicated in high-availability in Active/StandBy configuration sharing metadata via NFS to enable automatic or manual failover. In order to scale the name service horizontally,

is possible to split namespace into multiple federations with independent namenodes and namespaces. They do not require coordination with each other but use the same datanodes as common storage.

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks on datanodes. The size of the blocks is configurable by dfs.blocksize parameter. Every block is replicated as many times as specified by replication factor parameter (dfs.replication) according to a replica placement policy managed by active namenode. To realize data reliability, namenode needs to know network topology of the cluster, and so the node-rack relationship, to place file blocks on datanodes according to replica policy that by default writes the first replica on a node of the local rack, and second and third replica on different nodes of a remote rack, considering three as replication factor. The file-system resists the failure of a whole rack. In our activities, we developed two custom replica policies: One Replica Policy and Hierarchical Policy. The first one places a replica per rack in order to increase reliability (resisting the failure of two racks) and available bandwidth for read operation; the second one, instead, is able to exploit a geographically distributed infrastructure because it gives Hadoop the awareness of a hierarchical network topology organized in datacenters, racks and nodes. This data policy place first replica on a rack of a local site, and second and third on different racks of a remote site; in this way, the system resists failure of a whole datacenter. After a datanode failure, automatically Namenode schedules a re-replication of the blocks stored on that datanode. If it come back up, blocks are marked as over-replicated, so they will be deleted automatically in order to balance the number of replicas [Donvito et al. (2014)].

2.7 Simulation of large-scale distributed systems

This section will review the state of the art in simulation frameworks and tools for large-scale distributed systems, beginning with general-purpose simulation frameworks, continuing with network simulation, and finishing with the SimGrid simulation framework.

2.7.1 General purpose simulation frameworks

Parsec [Bagrodia et al. (1998)] is a simulation environment developed at UCLA that provides these features:

- An easy path for the migration of simulation models to operational software prototypes.
- Implementation on both distributed- and shared-memory platforms and support for a diverse set of parallel simulation protocols.
- Support for visual and hierarchical model design.

This environment consists of three primary components: a parallel simulation language called Parsec (parallel simulation environment for complex systems); its GUI, called Pave; and the portable runtime system that implements the simulation algorithms.

A simulation in Parsec consists of a series of events, which must be executed in the order of their time stamps. On a single processor, these events can be placed in a central queue so that the global event list algorithm can correctly order them. When run in parallel, however, not only

is the event list distributed so that each processor has only a portion of it, but events may also arrive asynchronously from other processors [Bagrodia et al. (1998)].

Parsec adopts the process-interaction approach to discrete-event simulation. A Parsec program consists of a set of entities and C functions. Each entity is an LP that models a corresponding physical process; entities can be created and destroyed dynamically. Events are modeled by message communications among the corresponding entities. Each message carries a logical time stamp matching the time at which the corresponding event occurs in the physical system. An entity may also schedule for itself a special message, called a timeout, for a specific time in the future. This message is often used by an entity to simulate the passage of time in the physical system, and its handling has been optimized in the system [Bagrodia et al. (1998)].

Parsec supports the following synchronization algorithms [Bagrodia et al. (1998)]:

- A sequential or global event-list algorithm.
- Three parallel conservative algorithms: a null- message-based algorithm, a conditional event algorithm, and a combination of the two (the accelerated null message (ANM) algorithm).
- An optimistic algorithm based on space-time simulations.
- The ideal simulation protocol (ISP), based on the critical path concept, which predicts a realistic lower bound on the execution time of a given parallel model.

OMNeT++ [Varga (2001), Varga and Hornig (2008), Varga (2010)] is a C++-based discrete event simulator for modeling communication networks, multiprocessors and other distributed or parallel systems. OMNeT++ represents a framework approach. Instead of directly providing simulation components for computer networks, queuing networks or other domains, it provides the basic machinery and tools to write such simulations. Specific application areas are supported by various simulation models and frameworks such as the Mobility Framework or the INET Framework [Varga (2013)]. These models are developed completely independently of OMNeT++, and follow their own release cycles. OMNeT++ was designed from the beginning to support network simulation on a large scale [Varga (2010)].

An OMNeT++ model consists of modules that communicate with message passing. The active modules are termed *simple modules*; they are written in C++, using the simulation class library. Simple modules can be grouped into *compound modules* and so forth; the number of hierarchy levels is not limited. Messages can be sent either via connections that span between modules or directly to their destination modules [Varga (2010)].

OMNeT++ also has support for parallel simulation execution. Very large simulations may benefit from the parallel distributed simulation (PDES) feature, either by getting speedup, or by distributing memory requirements. If the simulation requires several Gigabytes of memory, distributing it over a cluster may be the only way to run it. For getting speedup (and not actually slowdown, which is also easily possible), the hardware or cluster should have low latency and the model should have inherent parallelism. The communication layer is MPI, but it's actually configurable, so if the user does not have MPI it is still possible to run some basic tests over named pipes [Varga (2010)].

The user defines the structure of the model (the modules and their interconnection) in OMNeT++'s topology description language, NED. Typical ingredients of a NED description are

simple module declarations, compound module definitions and network definitions. Simple module declarations describe the interface of the module: gates and parameters. Compound module definitions consist of the declaration of the module's external interface (gates and parameters), and the definition of submodules and their interconnection. Network definitions are compound modules that qualify as self-contained simulation models [Varga (2010)].

The OMNeT++ package includes an Integrated Development Environment which contains a graphical editor using NED as its native file format; moreover, the editor can work with arbitrary, even hand-written NED code. The editor is a fully two-way tool, i.e. the user can edit the network topology either graphically or in NED source view, and switch between the two views at any time [Varga (2010)].

However, to effectively simulate a distributed system, specially a large-scale distributed system, the most critical part is the network. Next section describes the state of the art in network simulation, with an emphasis on efficient simulation of a network.

2.7.2 Network simulation

Section 2.7.2.1 presents general content on packet-level simulation, already presented in [Fujiwara and Casanova (2007)], and section 2.7.2.2 on flow-based simulation, already presented in [Casanova and Marchal (2002)] first, later in [Fujiwara and Casanova (2007)], and more in-depth in [Velho and Legrand (2009)], but included here for completeness.

This section will briefly review existing packet-level and flow-based simulators used by researchers in the area of grid computing. We then give some details about network simulation in SimGrid, since it is the simulation framework chosen for the evaluation of this thesis.

2.7.2.1 Packet-level simulation

Packet-level simulators use discrete-event simulation by which a flow over a network path can be represented as a sequence of events, such as packet arrivals and departures at end-points and routers. End-points and routers both implement full-fledge network protocols. Simulation time typically increases in proportion to the number of events [Liu et al. (2001)]. Popular such simulators include Cnet [McDonald (1991a,b)], SSFNet [Cowie et al. (1999b)], ns-1 [Bajaj et al. (1999)], ns-2 [The University of Southern California (2011)], ns-3 [Henderson et al. (2006, 2008), The NS-3 Consortium (2013)], GTNetS [Riley (2003)], and INET [Varga (2013)]. The main problem with these simulators is that simulation time can be orders of magnitude larger than simulated time for simulations that involve realistic topologies with many flows. For instance, using GTNetS, which is known for good scalability, simulating 200 flows each transferring 100MB between two random end-points in a random 200-node topology for 125 sec of simulated time takes approximately 1500 sec on a 3.2GHz Xeon processor [Fujiwara and Casanova (2007)]. While this is acceptable for researchers studying network protocols, it is prohibitive for many researchers studying distributed systems and algorithms on large-scale platforms for application that are long-running and/or that involve large amounts of communication. This problem is often compounded by the need to rely on results from over thousands of simulation experiments to compute valid statistics regarding the relative effectiveness of competing algorithms (see for instance the scheduling study in [N'Takpé and Suter (2006)], which uses over one million simulation experiments, with each experiments requiring over 1,000 sec of simulated time).

Several researchers have attempted to increase the speed of packet-level simulations. For instance, in [Liu and Chien (2003)] the authors developed the MaSSF framework, which combines the DaSSF packet-level simulator [Cowie et al. (1999a)] with message passing [Snir et al. (1998)] to accelerate and increase the scalability of network simulation by running in parallel on large clusters of workstations. MaSSF is the main component of the MicroGrid [Song et al. (2000)] tool for simulating Grid platforms and applications. Others have proposed emulation techniques by which traffic flows on physical devices, introducing delay, bandwidth and packet loss characteristics of the network to be simulated. A well-known example of such work is ModelNet [Vahdat et al. (2002)].

While the above works do increase the speed and scalability of packet-level simulation without compromising simulation accuracy, many users performing grid simulations need simulations orders of magnitude faster. Facing such requirements, simulators that relax the definition of a packet were developed. For instance, the Bricks simulator [Takefusa et al. (1999)] uses ideal queuing networks to simulate real networks. While the user can specify a packet size in this simulator, Bricks packets do not correspond to real network packets and Bricks does not implement real network protocols. Large packets lead to fast simulation but obviously low accuracy (in the extreme, multipath network communications use a store-and-forward approach with no pipelining across network links). Although lower packet size leads to behavior presumably qualitatively closer to that of real networks, nothing in this simulator ensure that the behavior is quantitatively close to that of, for instance, TCP. Another simulator, GridSim [Sulistio et al. (2007)] implements a protocol that includes some elements of UDP and allows for variable packet size. Like Bricks, GridSim requires small packet size to hope to gain accuracy close to that of true packet-level simulators on realistic network topologies, but then suffers from high simulation costs. Many other "grid" simulators exist, such as OptorSim [Cameron et al. (2004)], GangSim [Dumitrescu and Foster (2005)], Neko [Urbán et al. (2002)], or HyperSim [Phatanapherom et al. (2003)] (readers interested in depth details are invited to consult [Quetier and Cappello (2005)]). All implement some network model, but to the best of our knowledge (i.e., based on publications and/or on inspection of source codes), these simulators either use packet-level simulation or do not attempt to implement a model that realistically tracks the behavior of TCP networks.

2.7.2.2 Flow-based simulation

To increase the speed of network simulation one approach is to use theoretical models to compute the throughput of each flow in a network topology at a given time. Models have been proposed [Padhye et al. (1998), Mathis et al. (1997), Ott et al. (1997)] that model the throughput of a TCP flow as a function of packet loss and round trip delay, as well as some parameters of the network and of the TCP protocol. Unfortunately, some of these parameters are difficult to measure and/or instantiate for the purpose of grid simulations. Furthermore, it is not clear how the model can be applied to arbitrary network topologies with many simulated flows competing for network resources. Instead, one desires reasonable models that capture the bandwidth sharing behavior induced by TCP among flows on arbitrary topologies and that are defined by a few simple parameters, namely link physical bandwidths and TCP congestion window size. This notion of macroscopic models of bandwidth sharing is challenging [Massoulié and Roberts (2002)].

While it is important to model the throughput of a single flow appropriately, an equally important phenomenon to capture is the sharing of bandwidth among flows using the same link(s)

[Fujiwara (2007)]. Several researchers have explored the questions of bandwidth-sharing between TCP flows [Kelly (1997), Massoulié and Roberts (2002)]. Most works model bandwidth-sharing with fluid flows: flows are treated as continuous fluid rather than discrete packet instances.

The authors in [Casanova and Marchal (2002)] consider that the network is represented as a set of links $\mathcal{L}$ where link $l \in \mathcal{L}$ has a capacity $C_l > 0$. A flow is defined by a sequence of links, that is, a subset of $\mathcal{L}$. Let $\mathcal{F}$ be the set of flows. Let λ_f be the data transfer rate of flow f. A feasible bandwidth allocation must satisfy the following constraint:

$$\forall l \in \mathcal{L}, \sum_{f \ni l} \lambda_f \leqslant C_l \tag{2.1}$$

which states that links cannot deliver more bandwidth than their capacities. We now discuss three well-known models for bandwidth-sharing. We illustrate them on the classical example of a linear network, which is depicted in figure 2.4.

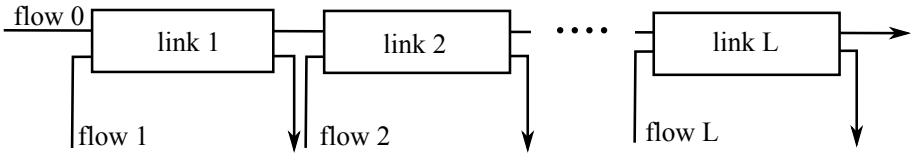


Figure 2.4: *Linear network with identical links of capacity C*

MaxMin Fairness MaxMin fairness is a traditional bandwidth-sharing principle [Bertsekas and Gallager (1992)]. The objective is to maximize the minimum of $\{\lambda_f\}$. λ_f is MaxMin fair if and only if an increase of any λ_f within the domain of feasible allocations must be at the cost of a decrease of some $\lambda_{f'}$ such that $\lambda_{f'} < \lambda_f$. This leads to the following formula:

$$\forall f \in \mathcal{F}, \exists l \in f, \sum_{f' \ni l} \lambda_{f'} = C_l \quad \text{and} \quad \lambda_f = max\{\lambda_{f'}, f' \ni l\} \tag{2.2}$$

In the linear network shown in Figure 2.4 under MaxMin fairness, all flows achieve the same data transfer rate:

$$\forall l, \ \lambda_l = C/2$$

Proportional Fairness Kelly [Kelly (1997)] questioned the validity of MaxMin fairness as a way to model TCP behavior. MaxMin fairness allocates more network resources to long flows than to short ones. Indeed, TCP is known to do just the opposite. As an alternative to MaxMin fairness, Kelly proposed proportional fairness, which is defined as follows. The objective of proportional fairness is to maximize

$$\sum_{\mathcal{F}} \lambda_f log(\lambda_f) \tag{2.3}$$

The solution must satisfy the following criteria: $\{\lambda_f\}_{f \in \mathcal{F}}$ is unique and for any other feasible allocation $\{\lambda'_f\}_{f \in \mathcal{F}}$, satisfies

$$\sum_{f\in\mathcal{F}} \frac{\lambda'_f - \lambda_f}{\lambda_f} \leqslant 0 \tag{2.4}$$

In the linear network shown in Figure 2.4 under proportional fairness, we find

$$\lambda_0 = \frac{C}{L+1}$$
$$\forall l \neq 0,\ \lambda_l = \frac{C(L-1)}{L+1}$$

Potential Delay Minimization Another idea is to minimize the time to complete all transfers. Assume that the data size transferred is fixed. One must then minimize the potential delay $1/\lambda_f$ for all flows. Such an allocation minimizes $\sum_{f\in\mathcal{F}} 1/\lambda_f$. In the linear network shown in Figure 2.4 under potential delay minimization, we find

$$\lambda_0 = \frac{C}{1+\sqrt{L}}$$
$$\forall l \neq 0,\ \lambda_l = \frac{C\sqrt{L}}{1+\sqrt{L}}$$

The question is then: which bandwidth-sharing principle holds for TCP connections on the Internet? The most widely known model is the simple MaxMin fairness model, which computes a bandwidth allocation in which increasing the allocation of a flow would require decreasing the allocation of another. However, it is well-known that TCP does not implement MaxMin fairness, as shown for instance by Chiu [Chiu (2000)]. The consensus is that TCP protocol is "close" to proportional fairness, since it favours short flows. However, Chiu also shows in [Chiu (2000)] that TCP does not implement proportional fairness exactly.

Indeed, the analytical models for TCP throughput in [Floyd and Fall (1999), Padhye et al. (1998)] approximate the throughput, $\lambda(p)$, to:

$$\lambda(p) = \frac{c}{RTT\sqrt{p}} \tag{2.5}$$

where p is the fraction of packets lost, RTT is the round-trip time, and c is some constant, provided that p is not "too high". Assuming that all flows experience the same loss rate, p, this formula suggests that bandwidth is in fact shared in inverse proportion to the RTT. This thus suggests a MaxMin scheme that is modified to account for low RTTs. Additionally, the TCP congestion mechanism relies on a window whose size is generally bounded (we denote by W this maximum window size), which impacts greatly the effective bandwidth of the flows (the effective throughput is thus bounded by W/RTT as there are always at most W pending packets).

Last, it has been proved that TCP sharing mechanism at the equilibrium is indeed equivalent to maximizing

$$\sum_i \frac{\sqrt{3/2}}{D_i} \tan^{-1}(\sqrt{3/2} D_i \lambda_i) \tag{2.6}$$

where D_i is the equilibrium round-trip-time [Low (2003)]. Such an equilibrium is generally different from the MaxMin sharing and should thus be more accurate. Solving such equations is however harder than the MaxMin sharing algorithm.

Based on the previous considerations, the designers of the SimGrid simulation tool [Casanova et al. (2008, 2014)], have opted for a RTT-aware MaxMin flow-level model. In this model, the bandwidths allocated to flows competing over a bottleneck link is inversely proportional to the flows' RTTs (a link is considered a bottleneck if the sum of the bandwidths allocated to the flows over this link is equal to the total bandwidth of the link), and the bandwidth of each flow is bounded by a value inversely proportional the inverse of its RTT.

$$\begin{aligned} &\text{Maximize } min_i\, RTT_i \cdot \lambda_f, \\ &\text{under constraints} \\ &\begin{cases} \forall l \in \mathcal{L}, \sum_{f|f \text{ uses } l} \lambda_f \leqslant C_l \\ \forall f \in \mathcal{F}, \lambda_f \leqslant \dfrac{W}{RTT_i} \end{cases} \end{aligned} \tag{2.7}$$

We refer the reader to [Casanova and Marchal (2002)] for full details on the model and for initial validation results via which this particular model was selected among several alternatives. The model is instantiated solely based on network link physical characteristics (latencies and bandwidths) and on the size of the TCP congestion window size. As a result, SimGrid is, to the best of our knowledge, the first simulation framework designed for the study of distributed systems and algorithms for large-scale platforms that uses a flow-level network simulation approach that attempts to capture the true behaviour of TCP networks and that decreases simulation costs by orders of magnitude when compared to packet-level simulation (simulations of more than 2 million nodes have been reported to work [Quinson et al. (2012)]).

2.7.3 SimGrid

The SimGrid framework [Casanova et al. (2008, 2014), The SimGrid Team (2014)] is a simulation-based framework for evaluating cluster, grid and P2P algorithms and heuristics.

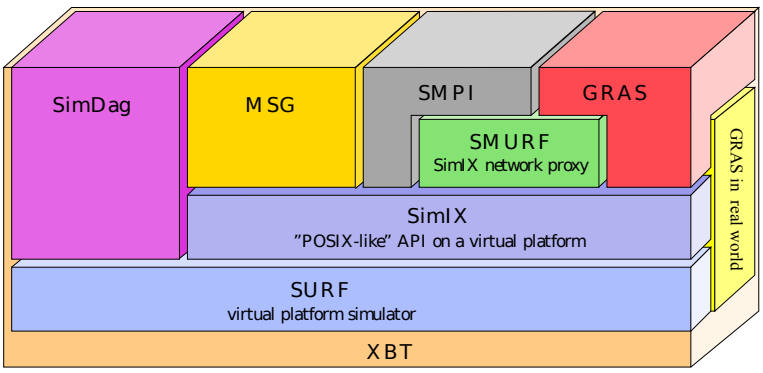


Figure 2.5: *SimGrid components overview*

SimGrid offers four user interfaces: SimDag, MSG, GRAS, and SMPI (see Figure 2.5). SimDag is the descendant of SimGrid v1 and is designed for the investigation of scheduling heuristics for applications as task graphs. MSG is the interface introduced in SimGrid v2 to study CSPs. GRAS allows to use SimGrid as a development lab for real distributed applications. SMPI enables the direct simulation of MPI applications.

XBT is a "toolbox" module used throughout the software, which is written in ANSI C for performance. It implements classical data containers, logging and exception mechanisms, and support for configuration and portability. SURF is the code-name of the simulation engine. SimIX is an internal module that provides a POSIX-like API on top of SURF, thus easing the development of simulation APIs that implement the abstraction of multiple concurrent processes. For instance, it would allow the development of openMP- or BSP-like user interfaces. The purpose of the SMURF module is to allow the distribution of simulated processes over a cluster, harnessing the memory of several computers. This would allow to improve the scalability of SimGrid, which is currently limited by memory.

The SimGrid simulation core [Donassolo et al. (2010)] implements and provides interfaces to a number of simulation models that vary in sophistication, and can be used to simulate different types of resources (network resources, computational resources). It consists of two main layers: the SURF layer implements the simulation models, and the SIMIX layer provides a low-level API to these models upon which user-level APIs can be developed. Both layers are described hereafter. Some simulation models share the same structure, which is implemented as an additional layer, called LMM, which is called by SURF and which is briefly described hereafter as well. MSG is one of the user interfaces that are offered to users. Users develop their algorithms on top of the MSG API. Figure 2.6 shows the whole picture of SimGrid.

MSG MetaSimGrid [Legrand and Lerouge (2002)], or MSG for short, is one of the four user interfaces of the SimGrid simulation framework. It provides some entities to model the participants in a simulation, and a set of functions to operate with. Next is the list of resource models in MSG:

Process Users need to simulate many independent scheduling decisions, so the concept of *process* is at the heart of the simulator. A *process* may be defined as a code, with some private data, executing in a *host*.

Host A *host* is any possible place where a *process* may run. Thus it may be represented as a physical resource with computing capabilities, some *mailboxes* to enable running *processes* to communicate with remote ones, and some private data that can be only accessed by local *processes*.

Task Since most scheduling algorithms rely on a concept of task that can be either computed locally or transferred on another processor, it seems to be the right level of abstraction for users purposes. A *task* may then be defined by a computing amount, a message size, and some private data.

Link Like in real-life environments, hosts are connected through network links. Then, a *link* represents the physical notion of a network link that connects two *hosts*, or a *host* with a switch. A *link* may then be defined by a latency, and a bandwidth.

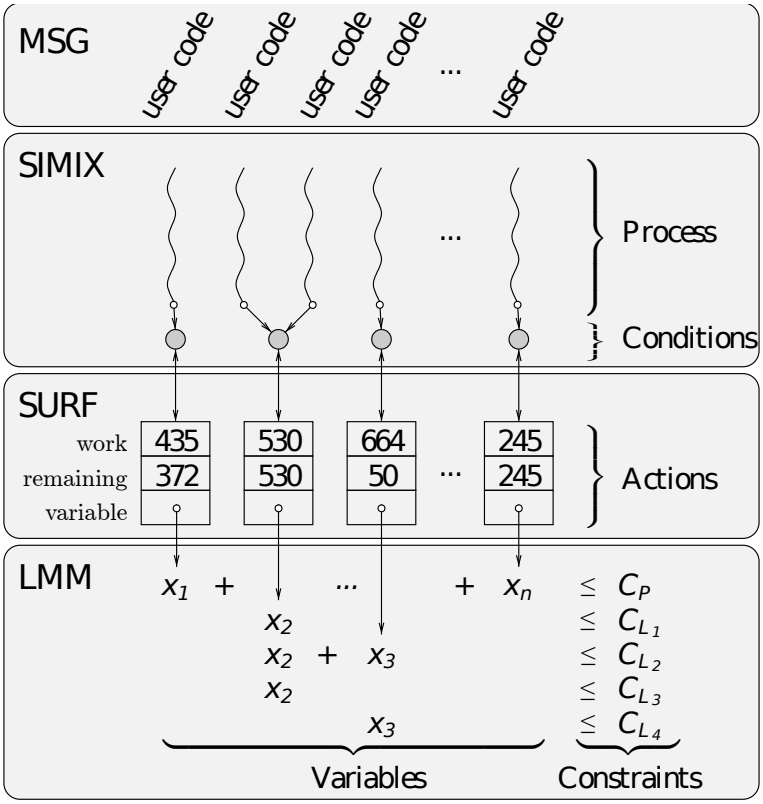


Figure 2.6: *SimGrid layers, and main data structures*

Mailbox For convenience, the simulator provides the notion of *mailbox* that is close to the TCP port notion.

Using the above entities a simulator should be described only in terms of *processes*, running on *hosts*, and interacting by sending, receiving, or processing *tasks*. Algorithms implemented on top of SimGrid should not have direct access to *links*, but rather should be implemented as a *process* that sends a *task* to a *host* using a *mailbox*. In fact, a *host* may have many *mailboxes*, and a *mailbox* is identified simply by a string. So, sending a *task* to a *host* using a *mailbox* amounts to transfer the *task* on a particular *link*, depending on the emitter *host*, and on the destination *host*, and to put it in a particular *mailbox*.

SIMIX SIMIX provides a Pthread-like API to manage concurrent simulated processes. More precisely, it provides the following abstractions: *processes*, *locks*, *condition variables*, and *actions*. Processes correspond to threads of control of the simulated application, locks and condition variables are used for synchronizing these threads of control, and actions are used to represent resource consumption generated by these threads of control. We briefly illustrate these abstractions via a simple example.

Consider a simulation of a computation on a host. This computation is embedded within a SIMIX process and launched via a user-level API call, e.g., `MSG_task_execute` if using the MSG API. This call creates a SIMIX action that corresponds to the amount of computation

to be performed (specified by the user-level API call). This action is associated with a SIMIX condition variable on which the process blocks. Once the action is completed, as dictated by the simulation models after some elapsed simulated time, the condition variable is signaled. The user-level API call returns control to the user, thereby providing the simulation of the delay incurred for performing the computation.

Most simulations consist of many SIMIX processes. All processes run in mutual exclusion and SIMIX is responsible for controlling their execution. Essentially, all processes run in round-robin fashion until all of them block on condition variables to wait for action completions. At a given simulated time t, SIMIX has thus a list of blocked processes. SIMIX then calls the lower layer of the simulation core, SURF, through the `surf_solve` function. SURF, discussed in the next section, is responsible for handling the simulation clock and the usage of simulated physical resources. `surf_solve` advances the simulation clock to time $t + \delta$ at which at least one of the actions waited upon has completed (or failed). A list of the completed (or failed) actions at time $t + \delta$ is returned to SIMIX. SIMIX then wakes up the corresponding processes. The same procedure is repeated, advancing the simulated time from task completion to task completion until all processes terminate.

The execution of the simulated application is handled by SIMIX, and is fully separated from the simulation of the underlying platform, which is handled by SURF. The two layers communicate solely via the condition variable and action abstractions, as shown in the top part of Figure 2.6.

SURF SURF provides several models for determining simulated action execution times and resource consumptions. These models can be selected and configured at runtime, and each model is responsible for actions and resources of a given type (e.g., CPU, network, timer). For instance, in terms of network resource models, the current implementation provides a default model of TCP networks [Velho and Legrand (2009)], a model that offloads all simulation to the GTNetS packet-level network simulator [Riley (2003)], a simple model based on uniform random distributions, and more advanced models that use Lagrangian optimization and gradient descent [Low (2003)]. By picking an appropriate model the user can trade off speed/scalability for accuracy, with no change to user source code.

All simulation models are accessed via the `surf_solve` function, which proceeds in the following steps:

1.- Query each active simulation model for the next action completion/failure date among all the actions managed by that model. This is done through the `share_resources` function, which each model must implement. For many models, this function relies on an extra layer, LMM, via which resource usage is represented as a set of linear constraints, as seen in the next section. This general approach enables to represent very complex situations. LMM uses a sparse representation of this linear system and uses a simple MaxMin allocation algorithm by default but also implements more sophisticated models based on the work in [Low (2003)]. As seen in Figure 2.6, the models in SURF keep track of the amount of work remaining for each action, and can therefore determine when each action will complete based on current simulated resource usage.

2.- Compute t_{min}, the minimum of these completion dates. Examine user-provided traces used to describe dynamically changing resource conditions to see whether a resource state change occurs before t_{min} (e.g., the available bandwidth of a network link increases, a host is

shutdown). If such a state change occurs, then call the `update_resource_state` function of the model in charge of the resource. Each model must implement `update_resource_state`. Update t_{min} to be the earliest time of next resource change.

3.- Ask each active model to advance the simulation time to t_{min} and to update every action state accordingly. This is done through the `update_action_state` function, which each model must implement.

4.- Return the set of actions that have finished or failed.

The LMM Layer Many of the simulation models in SimGrid represent actions and resources as variables and constraints in a linear system. For example, given a set $\mathcal{L}$ of network links defined by their bandwidths and a set $\mathcal{F}$ of network flows defined by the set of links they use, we can represent each flow f by a variable x_f (representing the bandwidth allocated to it). For each link l we have the following constraint:

$$\sum_{f \ni l} x_f \leqslant C_l \tag{2.8}$$

where C_l is the bandwidth of link l, which states that the bandwidth capacity of the link cannot be exceeded. For instance, in Figure 2.6, variable x_2 and x_3 correspond to two flows using respectively $\{L_1, L_2, L_3\}$ and $\{L_2, L_4\}$. Many allocations x can satisfy the set of link capacity constraints and different network protocols lead to different allocations [Low (2003)]. SimGrid uses a simple MaxMin allocation by default [Bertsekas and Gallager (1992)] but also implements more sophisticated models based on the work in [Low (2003)].

For such models, the LMM layer uses a sparse representation of the above constraints. The problem is solved by the `lmm_solve` function, which is efficient because its complexity is *linear* in the *system size*, where the system size depends on the number of actions, the number of active resources, and the complexity of the resource usage. For example, the system corresponding to a set of N CPUs running each an action would be of size $\Theta(N)$. The system corresponding to F flows going each through L links would be of size $\Theta(F \cdot L)$.

If the system needs to be modified it is invalidated and the allocation must be recomputed with possibly new variables and constraints. For example, in Figure 2.6, removing variable x_2 would force recomputation of variable x_3, removing variable x_1 would force recomputation of variable x_n, etc. More generally, such invalidations occur based on the action life-cycle (e.g., action creation, action termination, action suspension/resumption), i.e., between two successive calls to `surf_solve`, or based on resource state changes, i.e., when function `update_resource_state` is called. Although we have used network resources as an example, the same approach is applicable to other, arguably less challenging, resource types.

The Default CPU Model Like many other models, the default CPU model relies on a LMM system and associates each CPU with a constraint whose bound is the rate of the simulated CPU (in MFlop/s). We detail the components of this model along with their complexities:

Action creation An action is defined by its *remaining amount* of work (in MFlop), which is initialized upon creation, and by a corresponding variable in the LMM system. The resource

consumption rate allocated to the action varies over time depending on the value of this variable.

`share_resources` To compute the next action completion date, this function first computes a new solution of the LMM system, if needed. Then, it goes through the list of all active actions to compute when each would complete, based on its current resource share and remaining amount of work, assuming that the system remains unchanged. The complexity of this function is thus $\Theta(|\text{actions}|)$ plus possibly the complexity of `lmm_solve`, which is also $\Theta(|\text{actions}|)$.

`update_resource_state` When the state of a resource is changed, one needs only to update the bound of the corresponding constraint, which is done with complexity $\Theta(1)$.

`update_action_state` This function advances simulation time. To do so it goes through the list of all actions to update their remaining amounts of work, which leads to a $\Theta(|\text{actions}|)$ complexity.

The Default Network Model The communication time of a message for flow f is given by $T_f = S_f/\lambda_f + L_f$, where S_f is the message size, λ_f is the bandwidth allotted to f, and L_f is the sum of the latencies of the links traversed by f. However, according to [Velho et al. (2011)], L_f and B_l (used in the computation of λ_f) are physical characteristics that are not directly representative of what may be achieved by flows in practice. The protocol overhead should be accounted for, which can be done by multiplying all latencies by a factor $\alpha > 1$ and all bandwidths by a factor $\beta < 1$. α can account for TCP slow-start and stabilization, which prevent flows from instantaneously reaching steady-state. β can account for packing and control overheads. The above leads to that SimGrid communication time ($T_{L,B}^{(SG)}$) is well approximated by a linear function of message size (S) as follows:

$$T_{L,B}^{(SG)}(S) = \alpha \cdot L + \frac{S}{min(\beta \cdot B, \frac{W}{2L})} \tag{2.9}$$

where $W = 20000$ is the maximum window size, L the latency, and B the bandwidth.

The authors in [Velho and Legrand (2009)] propose the following model for SimGrid. Every link $\mathcal{L}_k$ has a maximum bandwidth B_k, and every flow $\mathcal{F}_i$ has a throughput ρ_i. Each flow $\mathcal{F}_i$ really starts after $\alpha \sum_{k|\mathcal{F}_i \text{ uses } \mathcal{L}_k} Lat_k$, and the bandwidth sharing of active flow is computed by solving the following program:

$$\begin{gathered}\text{Maximize } min_i\, \omega_i \cdot \rho_i, \\ \text{under constraints} \\ \begin{cases} \forall \mathcal{L}_k, \displaystyle\sum_{i|\mathcal{F}_i \text{ uses } \mathcal{L}_k} \rho_i \leqslant \beta \cdot B_k \\ \forall \mathcal{F}_i, \rho_i \leqslant \dfrac{W}{RTT_i} \end{cases}\end{gathered} \tag{2.10}$$

where

$$\omega_i = \sum_{k|\mathcal{F}_i \text{ uses } \mathcal{L}_k} \left(Lat_k + \frac{\sigma}{B_k} \right) \tag{2.11}$$

Research conducted in [Velho and Legrand (2009)] showed that the values that minimize the error are: $\alpha = 10.4$, $\beta = 0.92$, and $\sigma = 8775$. Later research conducted in [Velho et al. (2011)] suggests: $\alpha = 13.01$, $\beta = 0.97$, and $\sigma = 20537.14$. With the latter settings, the maximum error of SimGrid compared to GTNetS for messages whose sizes are $S \geqslant 100$KB is less that 10%, and the mean error decreases marginally to 3%.

2.8 Summary

This chapter has been presented a complete vision of distributed systems, distributed storage, and technologies used in large-scale distributed systems for data transfers.

Also, general purpose simulation frameworks models have been presented, including simulation of networks and networked environments.

Chapter 3

Proposal of a generic I/O architecture for large-scale distributed systems

This chapter explains the architecture and implementation of a parallel file system specifically suited for large-scale distributed systems. The rest of this chapter is organized as follows. First, the Motivation and objectives of this file system will be explained, then the Architecture of a generic I/O middleware for large-scale distributed systems is presented, and finally the Implementation of a parallel file system for large-scale distributed systems is detailed.

3.1 Motivation and objectives

This chapter aims to fulfill the first primary objective indicated in Section 1.2: to propose a **generic I/O middleware architecture for large-scale distributed systems**.

This section presents the motivation of using the parallel file system proposed as a viable solution for large-scale distributed systems, and the main use cases that need to be addressed, so that this parallel file system can be used as a generic I/O middleware for large-scale distributed systems.

3.1.1 Use cases of parallel file systems for large-scale distributed systems

The list of use cases of parallel file systems for large-scale distributed systems are:

- Use existing data servers through open protocols
- Connect clients and servers through the Internet
- Build parallel partitions over the Internet
- Build distributed partitions using full replicas

Use existing data servers through open protocols

Clusters and supercomputers typically use file systems and data servers optimized for high-speed networks. Parallel file systems for clusters and supercomputers are designed to work with different kinds of data servers. However, when a connection needs to traverse different administrative domains, as happens on the internet, NFS, PVFS, GPS, Lustre, etc. are not a feasible solution due to firewall rules prevent these kinds of systems to work properly. Thus, a set of open and standard protocols must be present in a generic I/O middleware for large-scale distributed systems, so that it can access to existing data servers on the internet (see Figure 3.1).

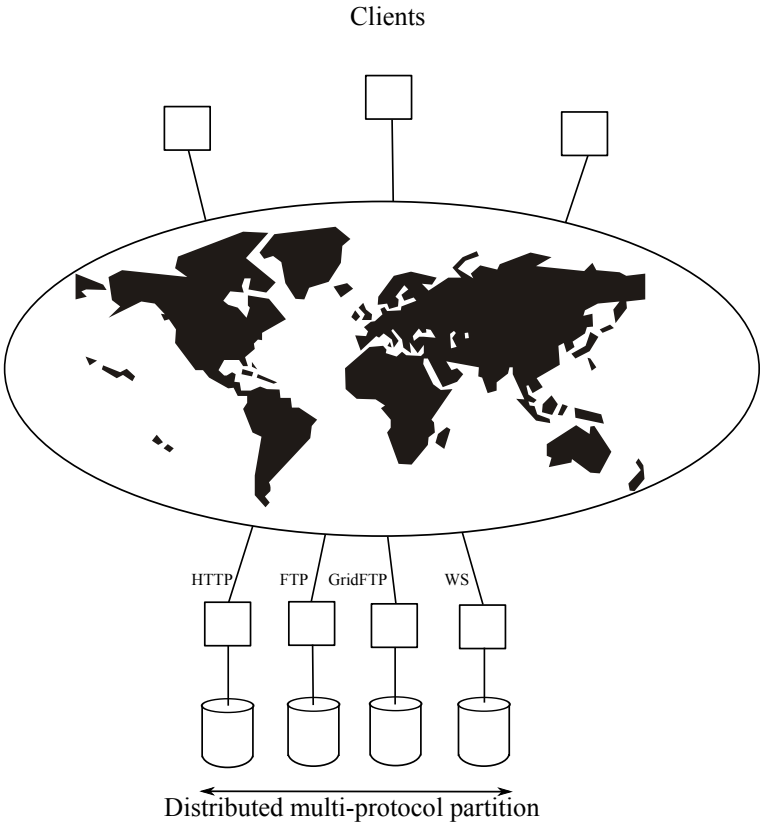


Figure 3.1: *Use case: Distributed multi-protocol partition*

Also, a generic I/O middleware must be able to use existing data servers without deploying further services. There are many examples of distributed file systems designed for big netowrks, Grids, and others. A very good example of these is Gfarm [Tatebe et al. (2002)]. However, these kinds of file systems need to deploy custom services in order to work. While this would be an ideal situation in terms of performance, it is only feasible in organizations under control, which does not happen when one needs to use existing servers running on the internet, and managed by third parties. Thus, a generic I/O middleware for large-scale distributed systems must be able to access those systems, and get the maximum benefit from those, without modifying the architecture, or deploying any further services.

Connect clients and servers through the Internet

In cluster environments and supercomputers nodes are typically connected through high-speed networks, like Gigabit Ethernet, 10 Gigabit Ethernet, Myrinet, Infiniband, optical fiber, etc. These networks have something in common, low latencies, and high bandwidth.

However, the communications that takes place on the internet have much higher latencies than on a high-speed network, and, in many cases, lower bandwidths also. A very good example of these two problems are DSL connections of final users, which have high latency, and asymmetric bandwidth (see Figure 3.2).

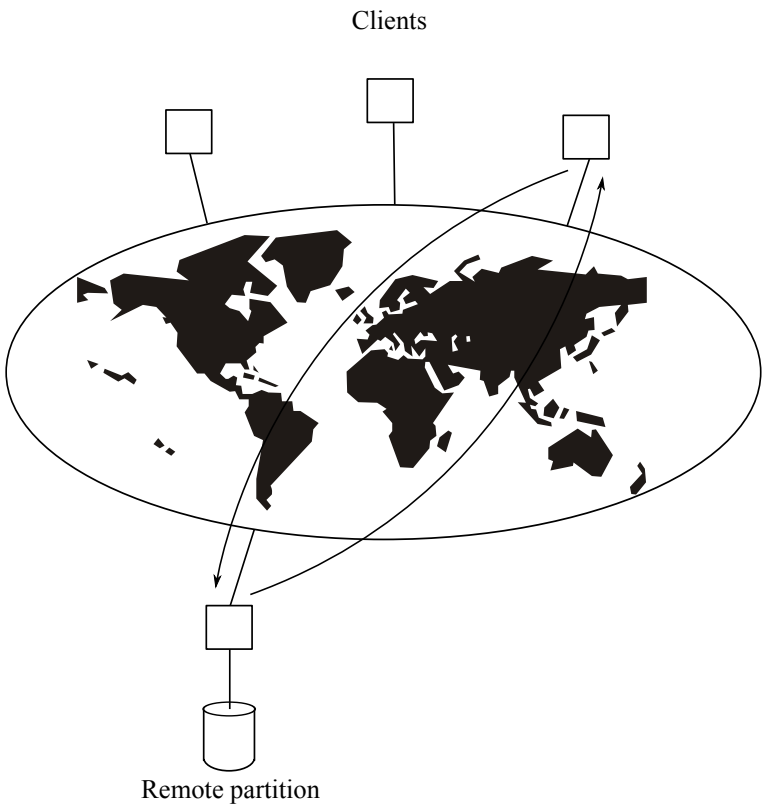


Figure 3.2: *Use case: Remote partition*

Build parallel partitions over the Internet

A generic I/O middleware for large-scale distributed systems must provide a parallel file system able to create and use parallel partitions. A parallel partition is that in which each data block is stored in a different server (see Figure 3.3). This kind of partition saves storage space while provides high throughput by parallel access to data servers. This case has two problems:

- The file system must work on a high-latency networks.
- It must be able to request certain blocks from the data server.

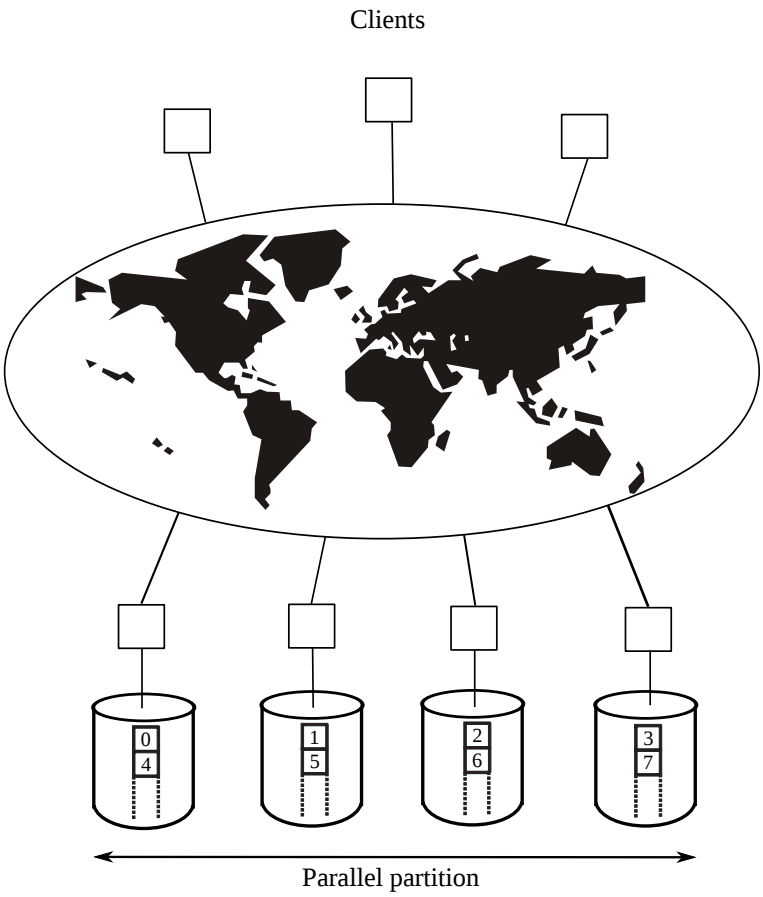


Figure 3.3: *Use case: Parallel partition*

Build distributed partitions using full replicas

A generic I/O middleware for large-scale distributed systems must be able to create and use replicas distributed on the internet. This means that this generic I/O middleware must be able to access to one or several full replicas. A fully replicated partition is that in which each data block of a file is stored in all servers that form the partition (see Figure 3.4). This kind of partition provides high throughput by parallel access to data servers, but it does not save storage space because every file is stored several times. However, it is a very common form of replicated content on the internet nowadays, since it is very easy to create a replica of a file. As in in the previous case, this has two problems:

- The file system must work on a high-latency networks.
- It must be able to request certain blocks from the data server.

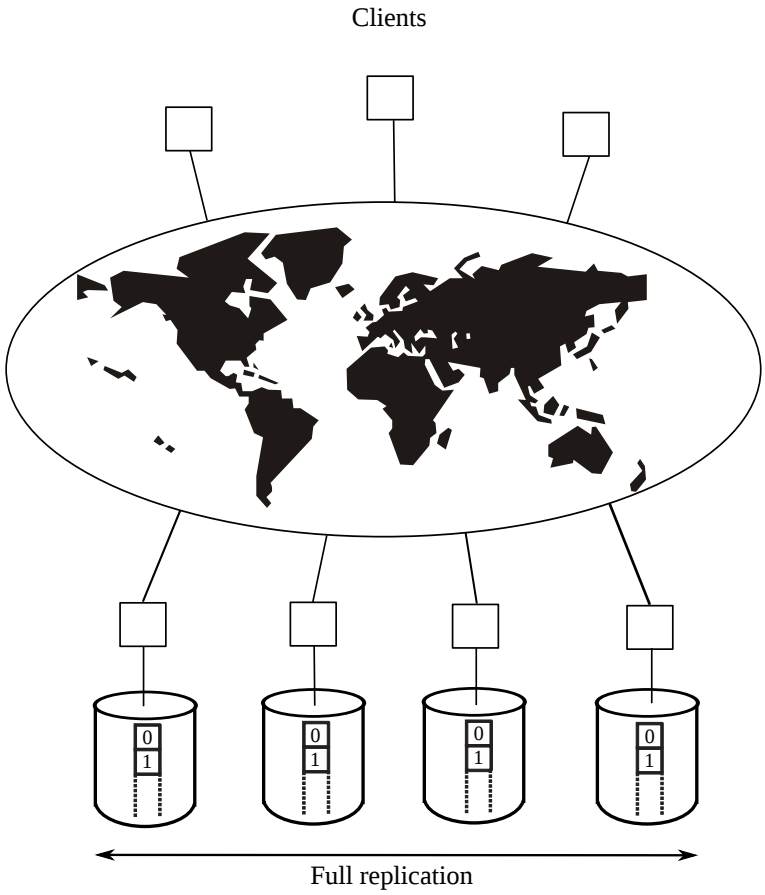


Figure 3.4: *Use case: Full replication*

3.2 Summary of the architecture, design, and implementation of *Expand*

This section summarizes the design and implementation of the Expand Parallel File System. An in-depth explanation of Expand can be found in Appendix A Architecture, design, and implementation of *Expand*.

Expand combines multiple NFS servers to create a distributed partition where files are striped. Expand requires no changes to the NFS servers and uses RPC operations to provide parallel access to the same file. Expand is also independent of the clients, because all operations are implemented using RPC and NFS protocols. Using this system, it can join heterogeneous servers (Linux, Solaris, Windows, etc.) to provide a parallel and distributed partition. This section describes the design and implementation of Expand.

The main motivation of the Expand design is to build a parallel file system for heterogeneous clusters using standard servers. To satisfy this goal, the authors designed and implemented a parallel file system using NFS servers.

The Network File system (NFS) [Sandberg et al. (1985)] supports the NFS protocol, a set of remote procedure calls (RPC) that provides the means for clients to perform operations on a

remote file server. This protocol is operating system independent. Developed originally for use in networks of UNIX systems, it is widely available in many systems, such as Linux or Windows, two operating systems very frequently used in clusters. Table 3.1 shows a simplified list of NFS server operations. In these operations, files are identified using file handles, an opaque structure that contains the information that the server needs to distinguish an individual file.

NFS operation	Description
lookup(fh, name) → (fh2, attr2)	Lookups the file name in a directory
create(fh, name, attr) → (fh2, attr2)	Creates a new file in a directory
remove(fh, name) → status	Removes a file from a directory
getattr(fh) → attr	Gets attributes associated to a file
setattr(fh, attr) → attr2	Sets attributes associated to a file
read(fh, offset, count) → (attr, data)	Reads data from file
write(fh, offset, count, data) → attr	Write data to a file
rename(fh, name, fh2, name2) → status	Modify the name of a file
link(fh2, name2, fh, name) → status	Add a new link
symlink(fh2, name2, string) → status	Add a new symbolic link
readlink(fh) → string	Gets name associated with a symbolic link
mkdir(fh, name, attr) → (fh2, attr2)	Create a new directory
rmdir(fh, name) → status	Removes a directory
readdir(fh, cookie, count) → entries	Returns entries information from a directory
statfs(fh) → status	Get file system information

Table 3.1: *Most significant NFS server operations*

Figure 3.5 shows the architecture of Expand. This figure shows how multiple NFS servers can be used as a single file system. File data are striped by Expand among all NFS servers, using blocks of different sizes as the striping unit. Processes in clients use an Expand library to access to an Expand distributed partition. Expand offers an interface based on POSIX system calls. This interface, however, is not appropriate for parallel applications using strided patterns with small access size [Nieuwejaar and Kotz (1996a)]. For parallel applications, Expand uses ROMIO [Thakur et al. (1999a)] to support the MPI-IO interface, implementing the appropriate Expand ADIO. This integration is explained in Section 3.2.5.

Using the former approach offers the following advantages:

1.- No changes to the NFS server are required to run Expand. All aspects of Expand operations are implemented on the clients.

2.- Expand is independent of the operating system used in the client. All operations are implemented using RPC and NFS protocols.

3.- Parallel file system construction is greatly simplified, because all operations are implemented on the clients. This approach is completely different to that used in many current parallel file systems, such as CFS [Pierce (1989)], Vesta [Corbett et al. (1993)], HFS [Krieger (1994)], PIOUS [Moyer and Sunderam (1994)], Scotch [Gibson (1995)], PPFS [Madhyastha (1997)], ParFiSys [Carretero et al. (1997), Pérez et al. (1997)], Galley [Nieuwejaar and Kotz (1996a,b, 1997)], and PVFS [Carns et al. (2000)].

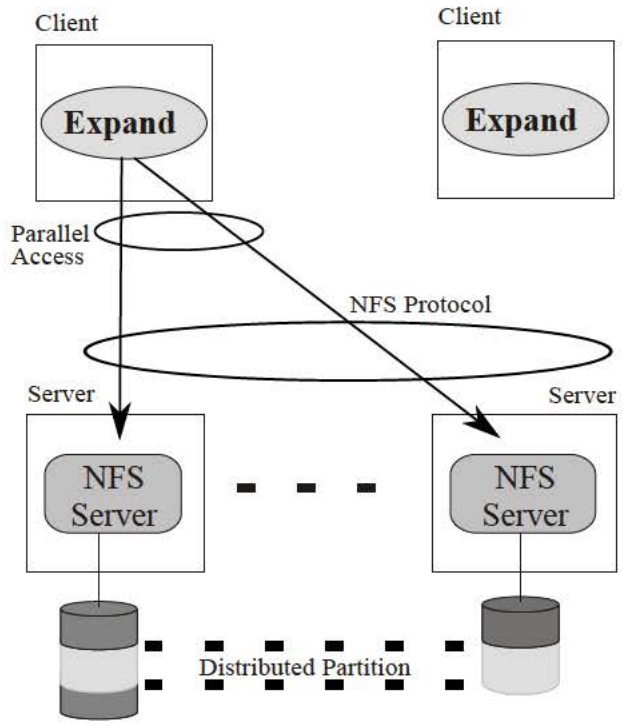


Figure 3.5: *Expand architecture*

4.- It allows the use of servers with different architectures and operating systems, because the NFS protocol hides those differences.

5.- It simplifies the configuration, because NFS is very familiar to users. A server only needs to export the appropriate directories, and clients only need a small configuration file that explains the distributed partition.

There are other systems that use NFS servers as the basis of their work, similarly to Expand. Bigfoot-NFS [Kim et al. (1994)], for example, also combines multiple NFS servers. However, this system uses files as the unit of interleaving (i.e., all data of a file reside in one server). Although files in a directory might be interleaved across several machines, it does not allow parallel access to the same file. Another similar system is the Slice file system [Chase et al. (2000)]. Slice is a storage system for high speed networks that uses a packet filter μproxy to virtualize the NFS, presenting to NFS clients a unified shared file volume. This system uses the μproxy to distribute file service requests across a server ensemble and it offers compatibility with file system clients. However, the μproxy can be a bottleneck that can affect the global scalability of the system.

The following sections describe data distribution, file structure, naming, metadata management, parallel access to files in Expand, access control and authentication, and, finally, ROMIO integration.

3.2.1 Data distribution and files

Expand combines several NFS servers (see Figure 3.6) in order to provide a generic distributed partition. Each server exports one or more directories that are combined to build a distributed

partition. All files in the system are striped across all NFS servers to facilitate parallel access, with each server storing a subfile of the parallel file conceptually. A file in Expand consists of several *subfiles*, one for each NFS partition. All subfiles are fully transparent to Expand users. On a distributed partition, the user can create, in the current prototype, *striped files with cyclic layout*. In these files, blocks are distributed across the partition following a round-robin pattern. This structure is shown in Figure 3.6.

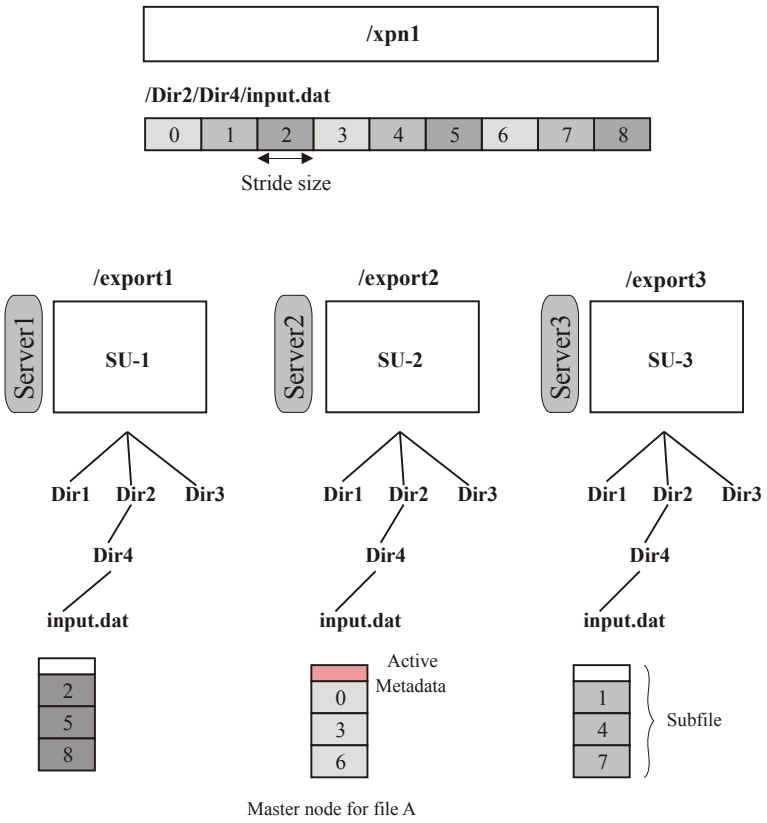


Figure 3.6: *File structure in Expand*

3.2.2 Naming and metadata management

Partitions in Expand are defined using a small configuration file. For example, the following configuration file defines two partitions:

```
/xpn1 8 4
server1 /export/home1
server2 /export/home2
server3 /export/home3
server4 /home

/xpn2 4 2
server1 /users
server3 /export/home/users
```

This configuration file defines two expand partitions. The first partition uses 4 servers (server1, server2, server3, and server4), and it uses by default a striping unit of 8 KB. The second partition uses two servers and it uses a striping unit of 4 KB. The path `/xpn1` is the root path for the first partition, and `/xpn2` is the root path for the second partition. So, the expand file `/xpn1/dir/data.txt` is mapped in the following subfiles:

```
/export/home1/dir/data.txt in server1
/export/home2/dir/data.txt in server2
/export/home3/dir/data.txt in server3
/home/dir/data.txt in server4
```

Each subfile of an Expand file (see Figure 3.6) has a small header at the beginning of the subfile. This header stores the file's metadata. This metadata includes the following information: *stride size*, *base node*, which identifies the NFS server where the first block of the file resides, and the *file distribution pattern* used. At present, it only uses files with a cyclic layout.

All subfiles have a header for metadata, although only one node, called the *master node* (described below) stores the current metadata. The master node can be different from the base node. To simplify the naming process and to reduce potential bottlenecks, Expand does not use any metadata manager, as used in PVFS [Carns et al. (2000)]. Figure 3.7 shows how directory mapping is performed in Expand. The Expand tree directory is replicated in all NFS servers. In this way, it can use the NFS lookup operation without any change to access to all subfiles of a file.

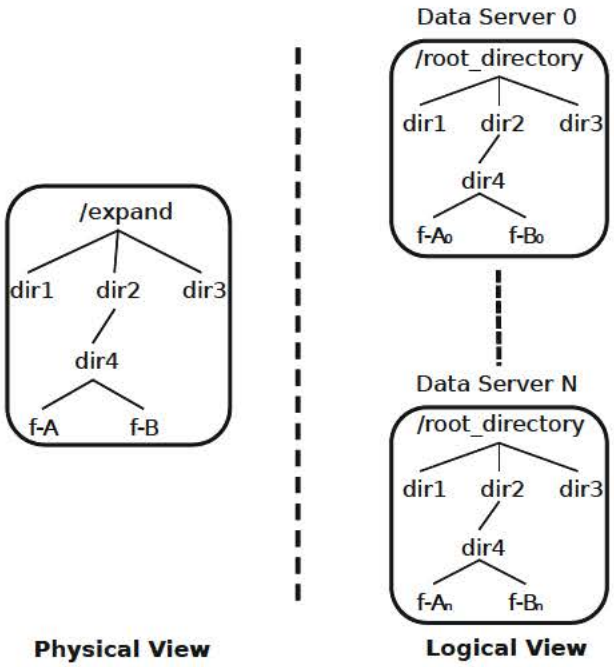


Figure 3.7: *directory mapping in Expand*

NFS clients use a *filehandle* to access the files. An NFS filehandle is an opaque reference to a file or directory that is independent of the filenane. All NFS operations use a filehandle to identify the file or directory which the operation applies to. Only the server can interpret

the data contained within the filehandle. Expand uses a *virtual filehandle*, which is defined as: $\bigcup_{i=1}^{N} filehandle_i$, where $filehandle_i$ is the filehandle used for NFS server i to reference subfile i belonging to the Expand file. The algorithm used in Expand for opening a file is very simple:

```
xpn_open(file) {
  Divide the file in (dir, name)
  for (i=0; i < numServers; i++) {
    Obtain the filehandle for dir in server_i (dfhi)
    lookup(dhi, name) -> fhi
  }
}
```

The metadata of a file resides in the header of a subfile stored in a NFS server. This NFS server is the *master node* of the file, similar to the mechanism used in the Vesta Parallel File System [Corbett et al. (1993)]. To obtain the master node of a file, the file name is hashed into the number of the node:

$$hash(filename) = NFS\ server_i \tag{3.1}$$

The hash function used in the current prototype is:

$$\left(\sum_{i=1}^{strlen(filename)} filename[i] \right) \ mod\ numServers \tag{3.2}$$

The use of this simple approach offers a good distribution of masters. Table 3.2 shows the distribution (standard deviation) of masters between several I/O nodes. These results have been obtained using a real file system with 145,300 files. The results shown in this table demonstrate that this simple scheme allows one to distribute the master nodes and the blocks between all NFS servers, balancing the use of all NFS servers and, hence, the I/O load.

Number of I/O nodes	Standard deviation
4	0.43
8	0.56
16	0.39
32	0.23
64	0.15
128	0.11

Table 3.2: *Distribution (standard deviation) of masters in different distributed partitions*

Because the determination of the master node is based on the file name, when a user renames a file, the master node for this file is changed. The algorithm used in Expand to rename a file is the following:

```
rename(oldname, newname) {
  oldmaster = hash(oldname)
  newmaster = hash(newname)
  move the metadata from oldmaster to newmaster
}
```

This process is shown in Figure 3.8. Moving the metadata is the only operation needed to maintain coherence for all the Expand files.

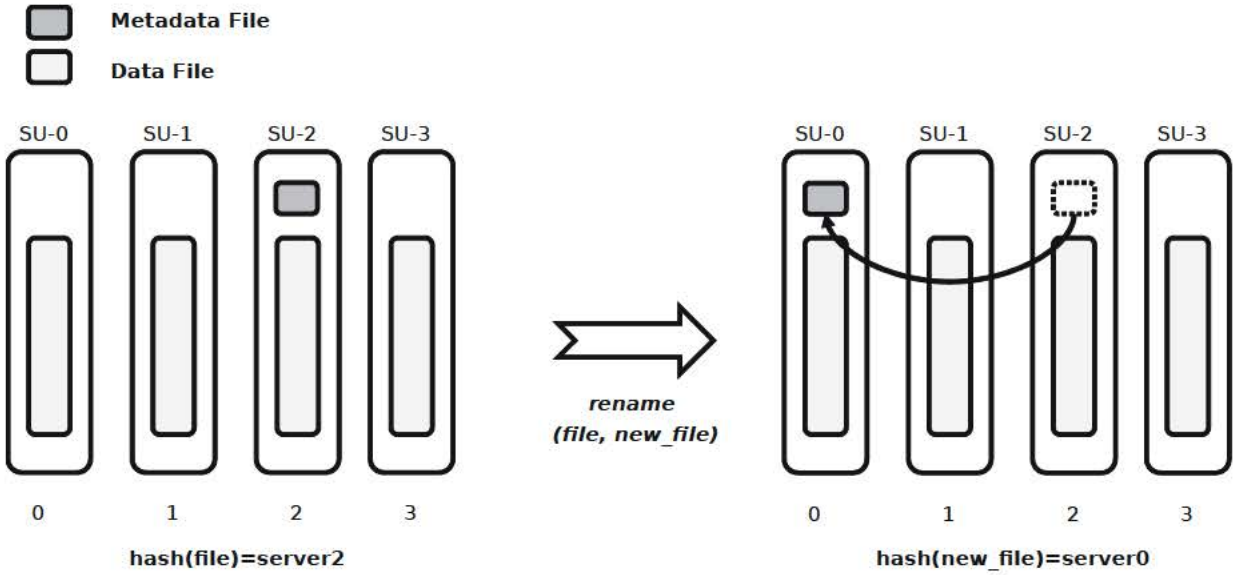


Figure 3.8: *Rename process in Expand*

3.2.3 Parallel access

All file operations in Expand use a *virtual filehandle*. This virtual filehandle is the reference used in Expand to reference all operations. When Expand needs to access a subfile, it uses the appropriated filehandle. Because filehandles are opaque to clients, Expand can use different NFS implementations for the same distributed partition.

To enhance I/O, user requests are split by the Expand library into parallel subrequests sent to the NFS servers involved. When a request involves k NFS servers, Expand issues k requests in parallel to the NFS servers, using threads to parallelize the operations. The same criteria are used in all Expand operations. A parallel operation to k servers is divided into k individual operations that use RPC and the NFS protocol to access the corresponding subfile. This process is shown in Figure 3.9.

3.2.4 Access control and authentication

The NFS server is stateless and does not keep files open on behalf of its clients, so the server must check the user's identity against the file access permission attributes on each request. The Sun

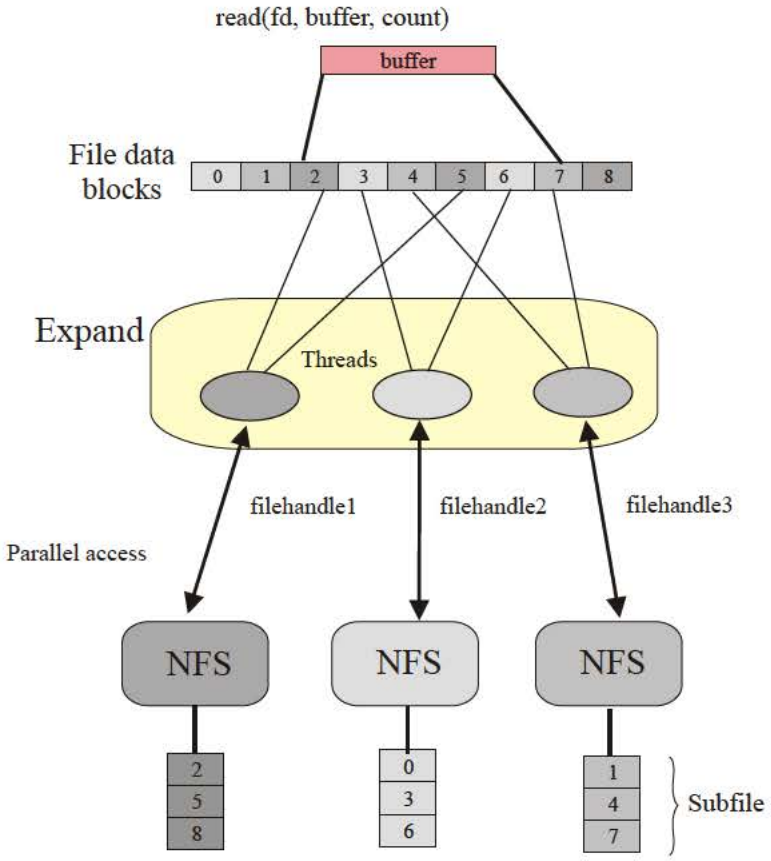


Figure 3.9: *Parallel access in Expand*

RPC protocol requires clients to send user authentication information with each request and this information is checked against the access permission in the file attributes. The current prototype of Expand uses `AUTH_SYS` authentication. With this authentication the server gets the client's UID and GID on each call and uses them to check permission. Using UID and GID implies that the client and server share the same UID assignments. This requirement for a consistent UID/GID mapping across many clients and servers can be resolved using the NIS name service.

3.2.5 User interface

Expand offers two different interfaces. The first inter- face is based on the POSIX system call. This interface, however, is not appropriate for parallel applications using strided patterns with small access size [Nieuwejaar and Kotz (1996a)]. Parallel applications can also use Expand with MPI-IO [Calderón et al. (2002a,b)]. Expand has been integrated inside ROMIO [Thakur et al. (1999a)] and can be used with MPICH (see Figure 3.10). Portability in ROMIO is achieved using an *abstract-device interface for IO* (ADIO).

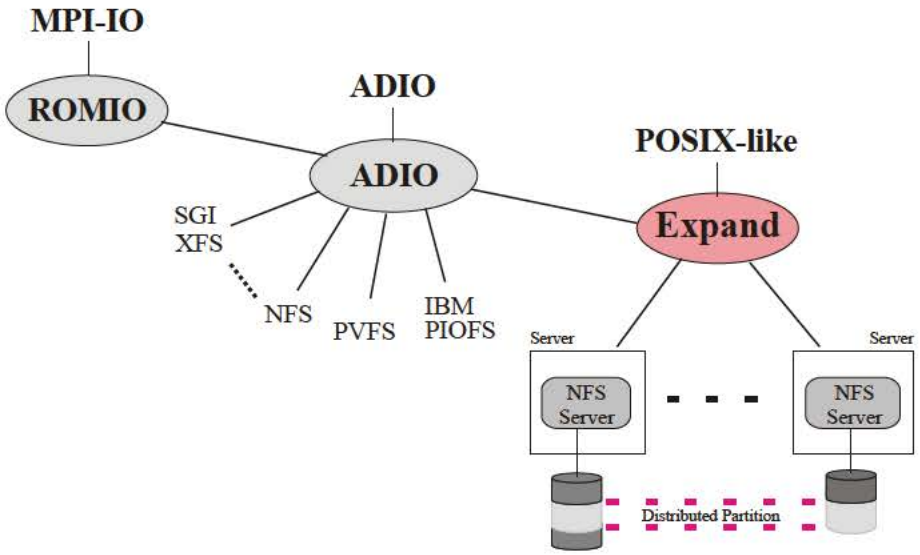


Figure 3.10: *Expand integration inside ROMIO*

3.2.5.1 ROMIO Integration

Figure 3.10 shows Expand integration inside ROMIO. ROMIO is a MPI-IO implementation using an abstract device input output (ADIO). ADIO [Thakur et al. (1999a)] is a mechanism specifically designed for implementing parallel I/O APIs portably on multiple file systems. ADIO consists of a small set of basic functions for parallel I/O. The ROMIO implementation of MPI-IO uses ADIO to achieve portability. The ADIO interface implemented for Expand includes:

- *Open a file.* All opens are considered to be collective operations.
- *Close a file.* The Close operation is also a collective operation.
- *Contiguous reads and writes.* ADIO provides separate routines for contiguous and noncontiguous access.
- *Noncontiguous reads and writes.* Parallel applications often need to read or write data that is located in a noncontiguous fashion in files and even in memory. ADIO provides routines for specifying noncontiguous accesses with a single call.
- *Nonblocking reads and writes.* ADIO provides nonblocking versions of all read and write calls.
- *Collective reads and writes.* A collective routine must be called by all processes in the group that opened the file.
- *Seek.* This function can be used to change the position of the individual file pointer.
- *Test and wait.* These operations are used to test the completion of nonblocking operations.
- *File Control.* This operation is used to set or get information about an open file.

- *Miscellaneous*. Other operations included in ADIO provide routines for deleting files, resizing files, flushing cached data to disks, and initializing and terminating ADIO.

Expand integration needs several phases:

1.- Implement the Expand-ADIO Interface (ADIOI).

2.- Connect Expand ADIO into ROMIO.

3.- Modify the MPICH compilation chain to include Expand ADIO in the compilation process.

3.3 Architecture of a generic I/O middleware for large-scale distributed systems

The proposed architecture for a generic I/O middleware is based on a different approach to the vast majority of parallel file systems:

- Use existing data servers through open protocols
- Connect clients and servers through the Internet
- Build parallel partitions over the Internet
- Build distributed partitions using full replicas

The parallel file system for large-scale distributed systems is designed according to the client/server model, where the client side of the file system is responsible for receiving requests from processes through different types of interfaces, such as *POSIX*, *FUSE*, or by command line tools; and the server side is formed by any standard networked file server that can operate through different administrative domains, and can typically traverse common firewall rules.

The client is responsible to contact the servers using the protocols associated with them. For this, the client is designed as shown in Figure 3.11, divided into four layers.

The top layer is responsible for providing various access interfaces to parallel applications. The layer named *core* is responsible for defining the basic algorithms used for data access. Inside this layer is located the module called *policy*, that is responsible for defining what are the actions to take in the operations of metadata location, selection of servers involved in every operation, etc. Both *core* and *policy* use the services of the layer called *NFI*, which stands for *Network File Interface*. This layer provides an interface to the basic operations of a file system. The lower layer is responsible for implementing the interface provided in the *NFI* layer for the different access protocols to a file system.

The data of a file are scattered by the generic I/O middleware through all servers using blocks of a certain size as distribution unit, or fully replicating the file across all the servers. The processes of a parallel application are the clients that use the generic I/O middleware library to access to a distributed partition.

The generic I/O middleware provides an interface based on POSIX-like calls. However, this interface is not enough for unmodified applications that use the system *POSIX* interface. For unmodified applications to work with the generic I/O middleware a *FUSE* interface is also needed.

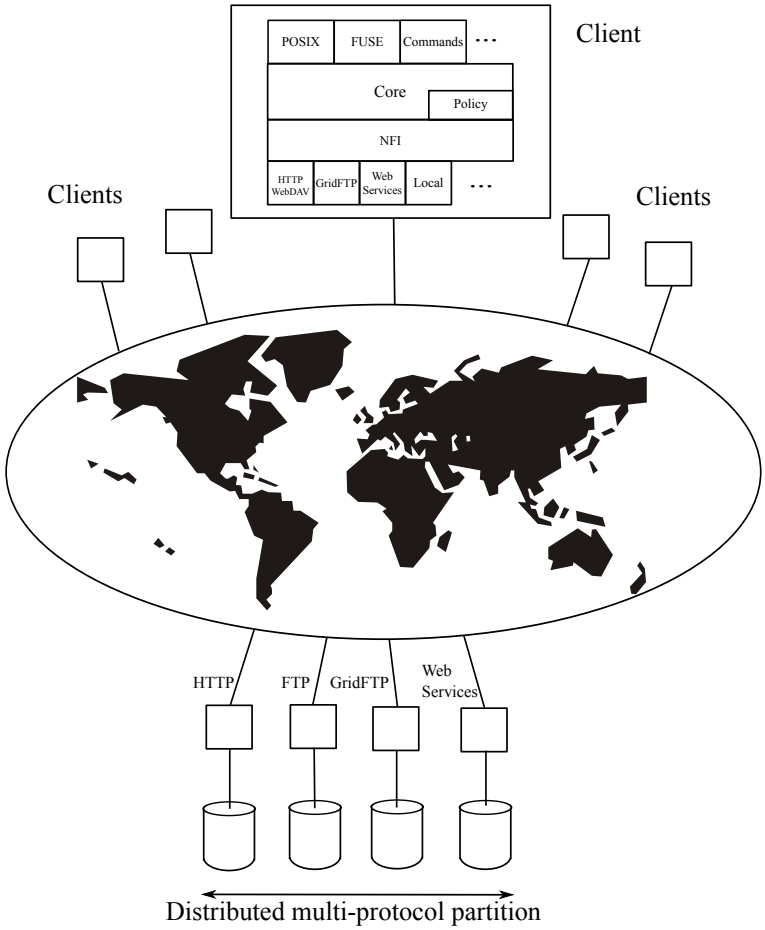


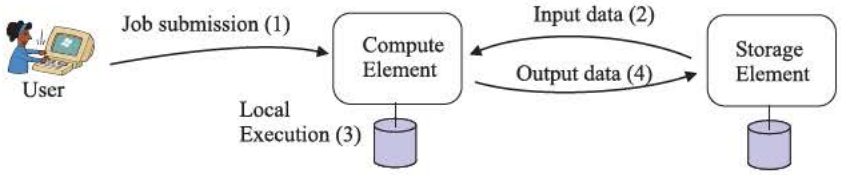
Figure 3.11: *Generic I/O architecture for large-scale distributed systems*

Using the outlined approach for the design of generic I/O middleware we have the following advantages:

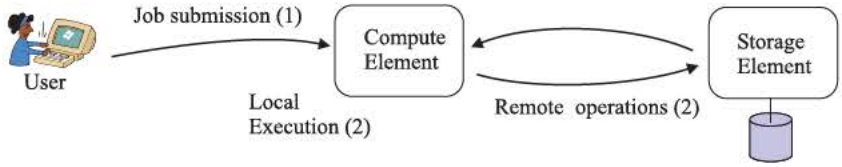
- No changes are required on the servers (HTTP, WebDAV, FTP, GridFTP, Web services, etc.) to benefit from the generic I/O middleware. All aspects of the operations are deployed in the client.
- The generic I/O middleware is independent of the operating system used on clients. All operations are implemented using standard protocols.
- It allows the use of servers running on different architectures and operating systems, since the use of standard protocols hides these differences.
- Building a file system is greatly simplified because all operations are implemented in clients.
- The configuration of the system is much simpler as standard servers, such as HTTP, are very familiar to users. The server only needs to export the appropriate directories, and clients only need a small configuration file detailing how is the distributed partition.

3.3.1 Remote I/O in distributed applications

This section shows the analytical model for executing applications in traditional distributed environments, where data files must be transferred to the computing node (see Figure 3.12(A)), compared with the model for executing applications using a parallel file system that provides remote I/O (see Figure 3.12(B)). The usage of such a file system avoids the *StageIn* and *StageOut* phases that appear when traditional approaches are used [Bergua et al. (2008)].



A) Traditional execution of grid applications



B) Execution of grid applications with remote access

Figure 3.12: *Classic vs Remote models*

The table 3.3 shows definitions and notations used by the model. In this model, we suppose that an application process an input file and obtain as a result a new file. We suppose that the applications made N I/O operations both for read and write of s bytes each one. This means that the input and output file size is $N \cdot s$.

Table 3.3: *Definitions and notations for data access model*

Variable	Description
B_d	Disk transfer rate
t_{seek}	Average disk seek time
t_{lat}	Average disk latency time
B_n	Network transfer rate
L_n	Network Latency
N	Number of I/O operations per file
s	Access size per I/O operation
D	Transfer Data size in *StageIn* and *StageOut*
T_{Exec}	Execution time

The time needed for executing an application in a traditional approach is defined as:

$$T_{TradExec} = T_{SendExecutable} + T_{StageIn} + T_{Processing} + T_{StageOut}$$

Where:

$$T_{StageIn} = \frac{Ns}{D}\left(L_n + \frac{D}{B_n}\right) + \frac{2Ns}{D}\left(t_{seek} + t_{lat} + \frac{D}{B_d}\right)$$
$$T_{StageOut} = \frac{Ns}{D}\left(L_n + \frac{D}{B_n}\right) + \frac{2Ns}{D}\left(t_{seek} + t_{lat} + \frac{D}{B_d}\right)$$
$$T_{Processing} = 2N\left(t_{seek} + t_{lat} + \frac{s}{B_d}\right) + T_{Exec}$$

According to the above expressions, the traditional approach is as follows:

$$\begin{aligned} T_{TradExec} = {} & T_{SendExecutable} + T_{Exec} + \\ & + 2N\left(t_{seek} + t_{lat} + \frac{s}{B_d}\right) + \\ & + \frac{2Ns}{D}\left(L_n + \frac{D}{B_n}\right) + \\ & + \frac{4Ns}{D}\left(t_{seek} + t_{lat} + \frac{D}{B_d}\right) \end{aligned}$$

When remote I/O is used for executing applications in distributed environments, the *StageIn* and *StageOut* phases are not required, because in this case, all I/O operations are remote. The time for executing the same application in this case is defined as:

$$\begin{aligned} T_{RemoteIO} = {} & T_{SendExecutable} + T_{Exec} + \\ & + 2N\left(t_{seek} + t_{lat} + \frac{s}{B_d}\right) + \\ & + 2N\left(L_n + \frac{s}{B_n}\right) \end{aligned}$$

We are interested in analyzing those cases where $T_{RemoteIO} < T_{TradExec}$. Simplifying the above expressions, we obtain:

$$
\begin{aligned}
2N\left(L_n + \frac{s}{B_n}\right) &< \frac{2Ns}{D}\left(L_n + \frac{D}{B_n}\right) + \frac{4Ns}{D}\left(t_{seek} + t_{lat} + \frac{D}{B_d}\right) \\
L_n + \frac{s}{B_n} &< \frac{s}{D}\left(L_n + \frac{D}{B_n}\right) + \frac{2s}{D}\left(t_{seek} + t_{lat} + \frac{D}{B_d}\right) \\
L_n + \frac{s}{B_n} &< \frac{s}{D}L_n + \frac{s}{B_n} + \frac{2s}{D}\left(t_{seek} + t_{lat} + \frac{D}{B_d}\right) \\
L_n &< \frac{s}{D}L_n + \frac{2s}{D}\left(t_{seek} + t_{lat} + \frac{D}{B_d}\right) \\
1 &< \frac{s}{D} + \frac{2s}{D}\left(\frac{t_{seek} + t_{lat} + \frac{D}{B_d}}{L_n}\right)
\end{aligned}
$$

Finally, we obtain that $T_{RemoteIO} < T_{TradExec}$ is verified when:

$$
\frac{s}{D}\left(1 + \frac{2(t_{seek} + t_{lat} + \frac{D}{B_d})}{L_n}\right) > 1
$$

As we can see, this expression depends on several factors: the application access size (s), the transfer size (D) used in the *StageIn* and *StageOut* phases, the network latency (L_n) and disk access times (t_{seek}, t_{lat}, and B_d).

Figure 3.13 shows in logarithmic scale the value of the expression:

$$
H = \frac{s}{D}\left(1 + \frac{2(t_{seek} + t_{lat} + \frac{D}{B_d})}{L_n}\right)
$$

for different network latencies and different $\frac{s}{D}$ values, assuming a conventional disk.

In figure 3.13 $T_{RemoteIO} < T_{TradExec}$ when $H > 1$. This is always true for short latencies. For higher latencies the performance of the remote access is better when $\frac{s}{D}$ gets closer to 1. Therefore, the best situation is to perform as few operations as possible. This means that a way to improve the behavior of parallel systems for large-scale distributed systems is to implement a buffered cache in the file system to get a $\frac{s}{D} >= 1$. In this scenario the performance of remote I/O will always be better [Bergua et al. (2008)].

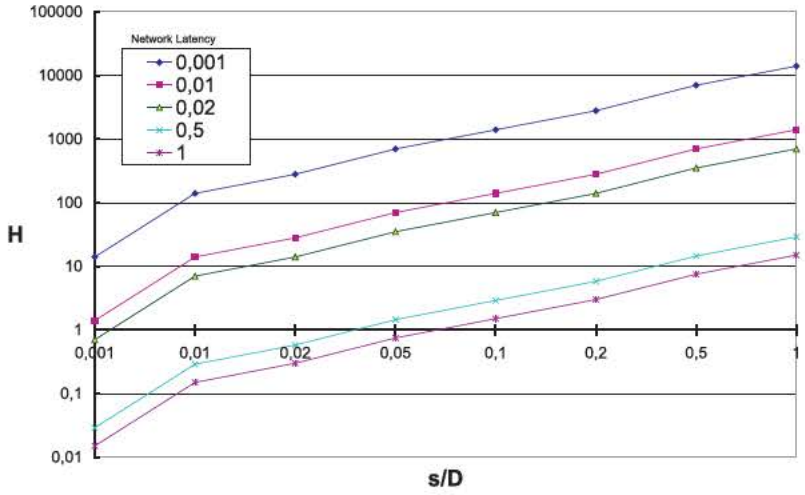


Figure 3.13: *H expression*

3.4 Implementation of a parallel file system for large-scale distributed systems

This section explains the implementation of the proposed architecture of a parallel file system specifically suited for large-scale distributed systems. To implement this architecture we have made used of *Expand*, the parallel file system for clusters and supercomputers described in chapter A, as the base file system for this proposal. In order to use *Expand* as a generic file system for large-scale distributed systems it needs several modifications and enhancements. This section details these modifications and enhancements made to *Expand* to adapt it to large-scale distributed systems, namely, Implementation of open protocols, Full replication or *mirroring* policy, Block grouping and reordering in *round-robin* policy, and Block grouping in full replication policy.

3.4.1 Problems of *Expand* for large-scale distributed systems

The main problems that *Expand* exhibits for being used in large-scale distributed systems are:

- *Expand* does not use open protocols for the internet
- *Expand* is not prepared for high-latency networks
- *Expand* does not use replicas, or any other distributed fault tolerance system

Expand does not use open protocols for the internet

Expand is a parallel file system that was originally designed for clusters. Thus, its first protocol that was implemented was NFS version 2, and later NFS versions 3 and 4 were incorporated since those are very usual file network file systems in clusters and supercomputers.

However, NFS and other network file systems designed for clusters and supercomputers cannot be used on the internet mainly for two reasons:

- They use ports that are usually filtered or firewalled when crossing the organization's boundaries.
- These file systems are designed assuming that they will run on low-latency high-bandwidth networks.

Therefore, *Expand* needs to incorporate new drivers for open protocols and services that are found on the internet nowadays, in large-scale systems, as HTTP, GridFTP, and web services.

Expand is not prepared for high-latency networks

Similarly as argued in the section above, *Expand* was designed for clusters. Clusters and supercomputing environments usually have low-latency high-bandwidth networks, as Gigabit Ethernet, 10 Gigabit Ethernet, Myrinet, Infiniband, optical fiber, etc.

However, a distributed application or file system that runs on the internet needs to traverse high-latency, and usually low-bandwidth also, networks. This problem requires to implement optimizations to *Expand*, so that access to data servers is made using as few requests as possible to minimize the side effect of high-latency networks.

Expand does not use replicas, or any other distributed fault tolerance system

Expand was initially designed as a parallel file system. This implies that data blocks, when using multiple servers, are stored cyclically among all available data servers in round-robin. This approach optimizes storage space while provides maximum throughput. However, the failure of one single data server makes the any data completely unrecoverable.

Thus, *Expand* needs to support some sort of fault tolerance for distributed systems, so that the failure of a node does not make the whole system unusable. Among all fault tolerance algorithms, it is worth noting a basic replica management system, because it very easy to implement, and currently very common on the internet. Many large-scale systems have replicated files as a form of fault tolerance. Therefore, *Expand* needs to be able to support the management of full replicas, at least.

3.4.2 Implementation of open protocols

Three open protocols have been chosen for their inclusion in *Expand*. For this task, three implementations have been developed in *Expand*:

- HTTP driver implementation
- GridFTP driver implementation
- OGSA ByteIO web service implementation

3.4.2.1 HTTP driver implementation

Nowadays, HTTP is the clear dominant data server on the internet. Many different services are served by HTTP servers. Thus, an HTTP driver has been added to *Expand*. To implement the

HTTP we have used *FuseDAV*[1] [2] rev. c20061002001751, a FUSE module for mounting WebDAV shares, which has been patched to support remote access and operations in memory (since it uses cached files on a local dir). This *FuseDAV* module makes use of the *libneon* library[3] rev. r1801, a client library for managing HTTP and WebDAV connections, which has also been patched to support ranged PUTs[4].

3.4.2.2 GridFTP driver implementation

In Grid environments the GridFTP protocol is the standard for file tranfers. There was a partial implementation of the GridFTP protocol by means of the Globus eXtensible Input Output library (XIO) API[5]. The implementation was developed in [Bergua Guerra (2006), Bergua et al. (2007)], and later studied in depth in [García Carballeira et al. (2007)]. However, the Globus XIO GridFTP Driver[6] is a very limited interface. It barely provides functions for opening/closing, reading/writing files, and handle/attribute manipulation:

- globus_xio_open
- globus_xio_close
- globus_xio_read
- globus_xio_write
- globus_xio_handle_cntl
- globus_xio_attr_cntl

Thus, a complete GridFTP driver has been implemented in *Expand*. The GridFTP driver implemented uses the Globus API for GridFTP, which is a complete interface that offers a range of functions for file and directory management.

3.4.2.3 OGSA ByteIO web service implementation

The OGSA ByteIO Working Group proposes seven POSIX-like functions [Morgan (2006), Chue Hong et al. (2009)] for the Byte I/O interface (which is described in UML):

- Read
- Write
- Append
- TruncAppend

[1] http://0pointer.de/lennart/projects/fusedav/
[2] https://github.com/AndyA/FuseDAV
[3] http://www.webdav.org/neon/
[4] http://www.mail-archive.com/neon@webdav.org/msg00465.html
[5] http://www.globus.org/api/c/globus_xio/html/index.html
[6] http://www.globus.org/api/c-globus-5.0.3/globus_xio_gridftp_driver/html/index.html

- SeekRead
- SeekWrite
- GetResourceProperty

It is worth noting that this interface implies a session-less communication semantic, but that this design does not prohibit session capable client semantics. In fact, we assume that a number of common file interface APIs will be implemented in client libraries to provide convenient mechanisms for data access (for example, true POSIX-style functions, C++/Java/C# streams, etc.).

3.4.3 Full replication or *mirroring* policy

The most basic replication scheme for a file is replicating the whole file. This full replication or *mirroring* replication scheme is also known as RAID 1 in the context of storage drives. This replication scheme is very common on the internet due to its simplicity. Figure 3.14 shows an example of block access with four replicas in a fully replicated partition. A full replication policy has been implemented in *Expand*, so that it can make use of existing replicas in data servers to offer transparent parallel and remote access to users.

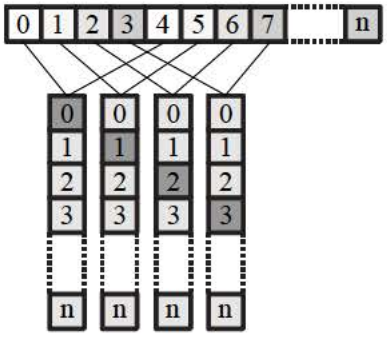


Figure 3.14: *Example of four replicas in a fully replicated partition*

3.4.4 Block grouping and reordering in *round-robin* policy

The default *round-robin* policy management in *Expand* is a cyclic policy. This means that each data block requested is retrieved from a different data server in round-robin, making one request per data block (see Algorithm 3.1).

```
1  roundrobin_no_grouping (fd, buffer, size, offset, num_servers)
2  {
3      new_offset = offset
4      count = 0
5
6      while (size > count) {
7          server = ((offset/block_size)%num_servers) + first_node)%num_servers
8          local_offset = (offset/block_size / num_servers * block_size) + (offset%block_size)
```

```
9
10        // local_size is the remaining bytes from new_offset until
              the end of the block
11        local_size = block_size − (new_offset%block_size)
12
13        // If local_size > the remaining bytes to read/write, then
              adjust local_size
14        if ((size − count) < local_size)
15            local_size = size − count
16
17        retrieve_block(buffer + count, server, local_offset, local_size)
18
19        count = local_size + count
20        new_offset = offset + count
21    }
22 }
```

Algorithm 3.1: *Round-robin with no block-grouping operation in Expand*

This process is depicted in Figure 3.15.

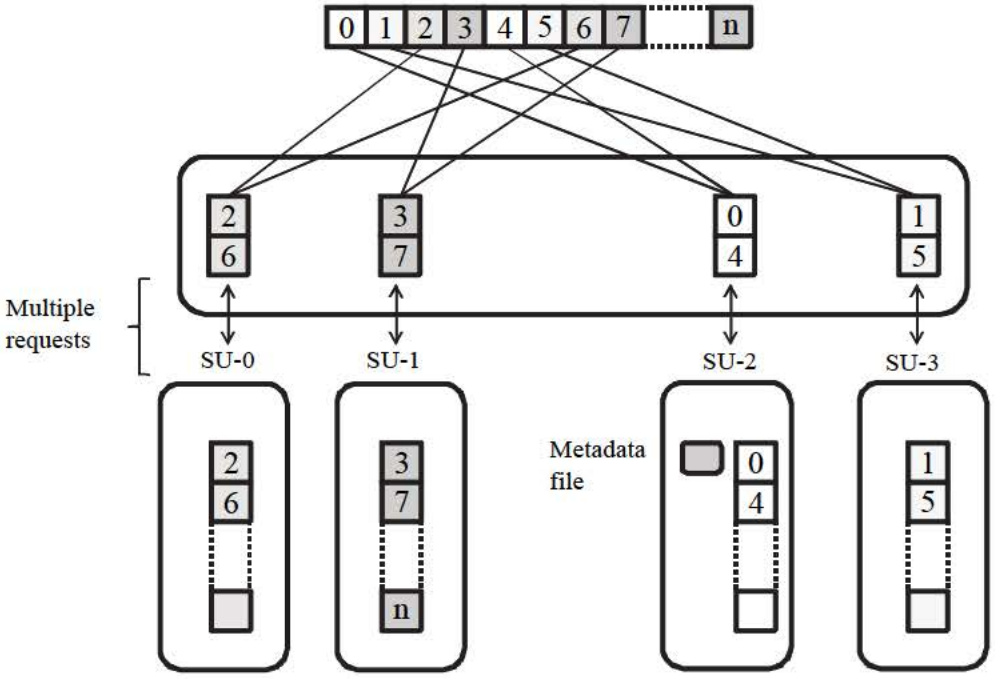


Figure 3.15: *Round-robin with no block grouping and reordering*

However, this process is not optimal, because if several data blocks need to be retrieved from the same server, *Expand* makes several requests, one for each data block. This process posed no problem in clusters or supercomputers due to the fact that in these environments nodes are connected through high-speed networks [Calderón (2005), Sánchez García (2009)]. However, in a large-scale system, distributed through different cities, or even countries, this can be a big problem mainly due to the high latencies typically observed between distant nodes on the internet. To

minimize the side effects of high latencies between nodes, a grouping and reordering optimization, that can be seen in Figure 3.16, has been incorporated to *Expand*. This optimization consists of three steps:

1.- First, all the data blocks that need to be retrieved from a data server are grouped (see Algorithm 3.2).

```
1  roundrobin_grouping (fd, buffer, size, offset, num_servers)
2  {
3      new_offset = offset
4      count = 0
5
6      while (size > count) {
7          server = ((offset/block_size %num_servers) + first_node)%num_servers
8          local_offset = ((offset/block_size)/num_servers * block_size) + (offset%block_size)
9
10         // local_size is the remaining bytes from new_offset
               until the end of the block
11         local_size = block_size - (new_offset%block_size)
12
13         // If local_size > the remaining bytes to read/write,
               then adjust local_size
14         if ((size - count) < local_size)
15             local_size = size - count
16
17         queue_block(io_queue[server],
18                         buffer + count, local_offset, local_size)
19
20         count = local_size + count
21         new_offset = offset + count
22     }
```

Algorithm 3.2: *Round-robin with block grouping operation in Expand*

2.- Then, all the blocks that need to be retrieved from a server are requested in one single call per server (see Algorithm 3.3).

```
1  roundrobin_retrieve (io_queue, contiguous_buffer)
2  {
3      count = 0
4      ∀ server ∈ io_queue:
5          local_offset = min_{io∈io_queue[server]}{io.offset}
6          local_size = Σ_{io∈io_queue[server]} io.size
7          retrieve_block(contiguous_buffer + count,
8                              server, local_offset, local_size)
9          count = count + local_size
10 }
```

Algorithm 3.3: *Round-robin retrieve operation in Expand*

3.- Finally, once all the necessary blocks are retrieved, they need to be reordered because contiguous blocks in a server do not preserve the right order. Thus, data blocks are reordered, so that each block is delivered to the user in its right place (see Algorithm 3.4).

```
1  roundrobin_reordering (fd, buffer, size, offset, num_servers,
       contiguous_buffer)
2  {
3      count = 0
4      ∀ server ∈ io_queue:
5          ∀ io ∈ io_queue[server]:
6              memcpy(io.buffer, contiguous_buffer + count, io.size)
7              count = count + io.size
8  }
```

Algorithm 3.4: *Round-robin block reordering operation in Expand*

This process is shown in Figure 3.16.

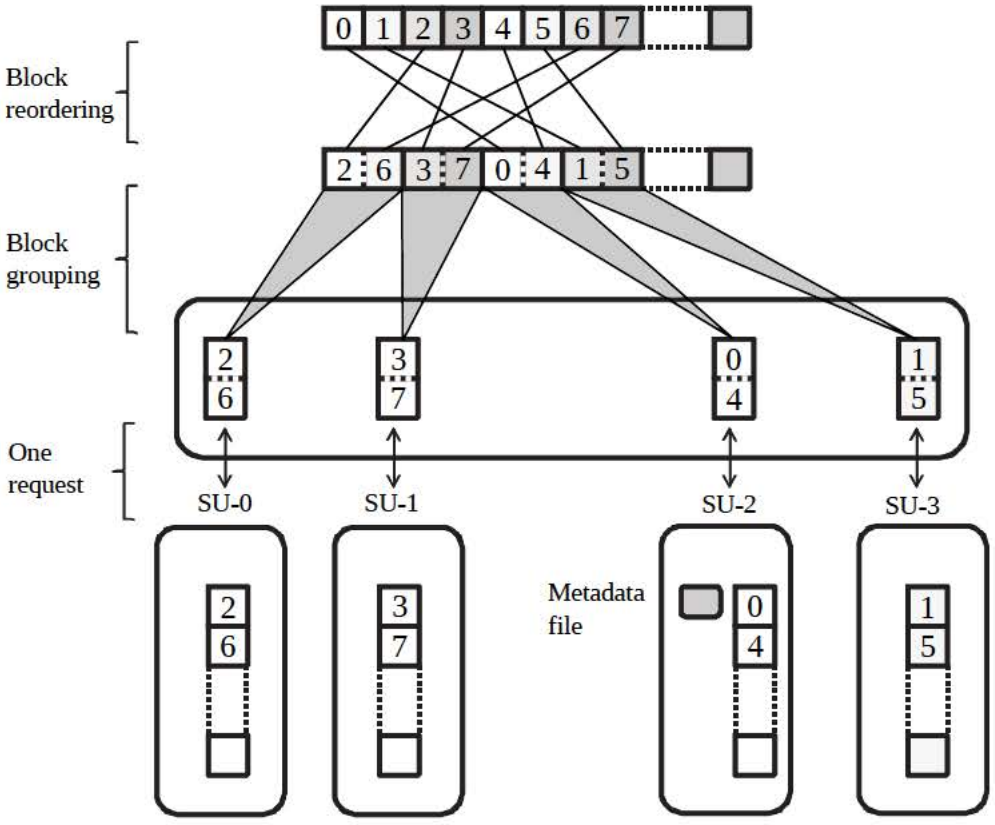


Figure 3.16: *Round-robin with block grouping and reordering*

3.4.5 Block grouping in full replication policy

Similarly to the *round-robin* case, the default full replication policy management in *Expand* is cyclic, what means that to retrieve each data block, a different request to each server is made (see Algorithm 3.5).

```
1  fullreplication_no_grouping (fd, buffer, size, offset, num_servers
       )
2  {
3      new_offset = offset
4      count = 0
5
6      while (size > count) {
7          server = ((offset / block_size) %num_servers) + first_node)%num_servers
8          local_offset = offset
9
10         // local_size is the remaining bytes from new_offset until
               the end of the block
11         local_size = block_size - (new_offset%block_size)
12
13         // If local_size > the remaining bytes to read/write, then
               adjust local_size
14         if ((size - count) < local_size)
15             local_size = size - count
16
17         retrieve_block(buffer + count, server, local_offset, local_size)
18
19         count = local_size + count
20         new_offset = offset + count
21     }
22 }
```

Algorithm 3.5: *Full replication with no block-grouping operation in Expand*

This process is depicted in Figure 3.17.

As argued in the *round-robin* case, this process in not optimal in the presence of high latencies. As done in *round-robin* policy, a grouping optimization has been incorporated to *Expand*. But in this case, there is no need of reordering, since in a fully replicated partition, every server stores the whole file, so we only need to select block ranges from each server because contiguous blocks hold the proper order. This optimization consists of two steps:

1.- First, all the data blocks that need to be retrieved from a data server are grouped (see Algorithm 3.6).

```
1  fullreplication_grouping (fd, buffer, size, offset)
2  {
3      l_size = size / ||servers||
4      for (i = 0 ; i < ||servers|| ; i++)
5      {
6          server = servers[i]
```

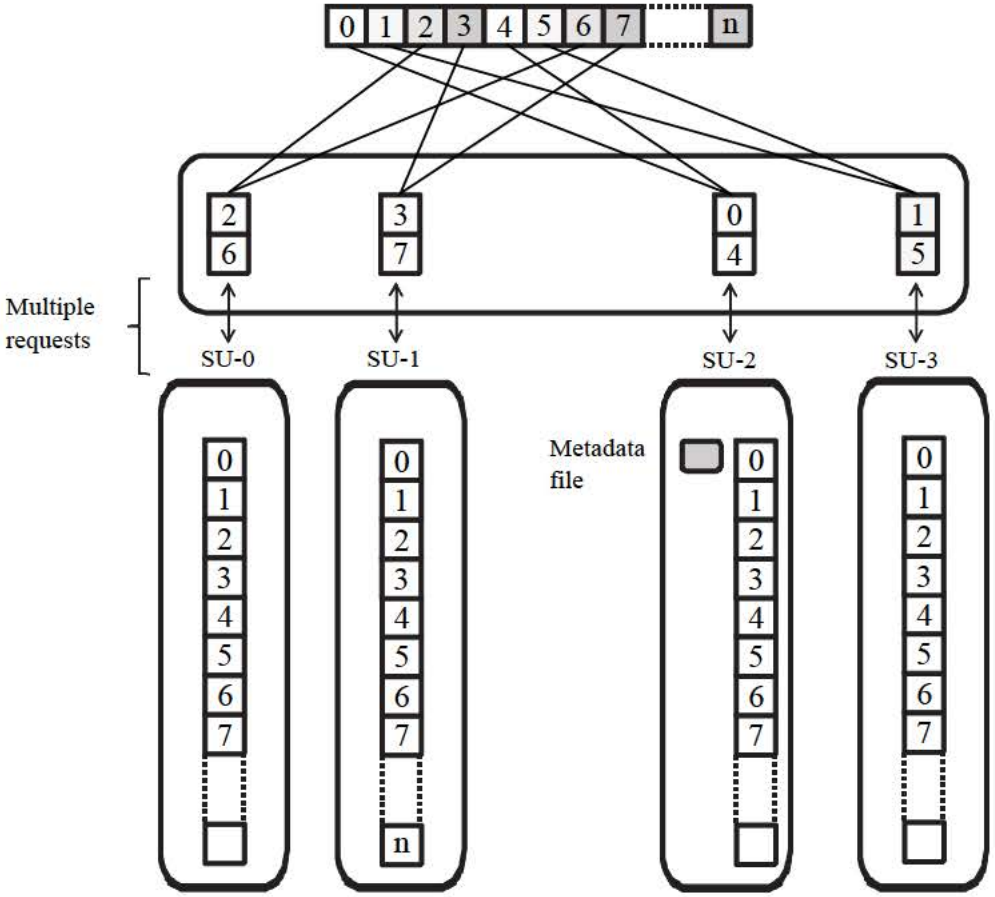


Figure 3.17: *Full replication with no block grouping*

```
7           local_offset = offset + i * l_size
8           local_size = l_size
9           local_buffer = buffer + i * l_size
10
11          queue_block(io_queue, server, local_buffer, local_offset
                , local_size)
12      }
13
14      if (size%||servers|| > 0)
15          io_queue[||servers|| − 1].size += size%||servers||
16      }
```

Algorithm 3.6: *Full replication block grouping operation in Expand*

2.- Finally, all the blocks that need to be retrieved from a server are requested in one single call per server (see Algorithm 3.7).

```
1   fullreplication_retrieve (io_queue, buffer)
2   {
```

```
3       count = 0
4
5       ∀ server ∈ io_queue
6       {
7           retrieve_block(buffer + count, server,
8                          io_queue[server].offset,
9                          io_queue[server].size)
10
11          count = count + io_queue[server].size
12      }
13  }
```

Algorithm 3.7: *Full replication retrieve operation in Expand*

This process is shown if Figure 3.18.

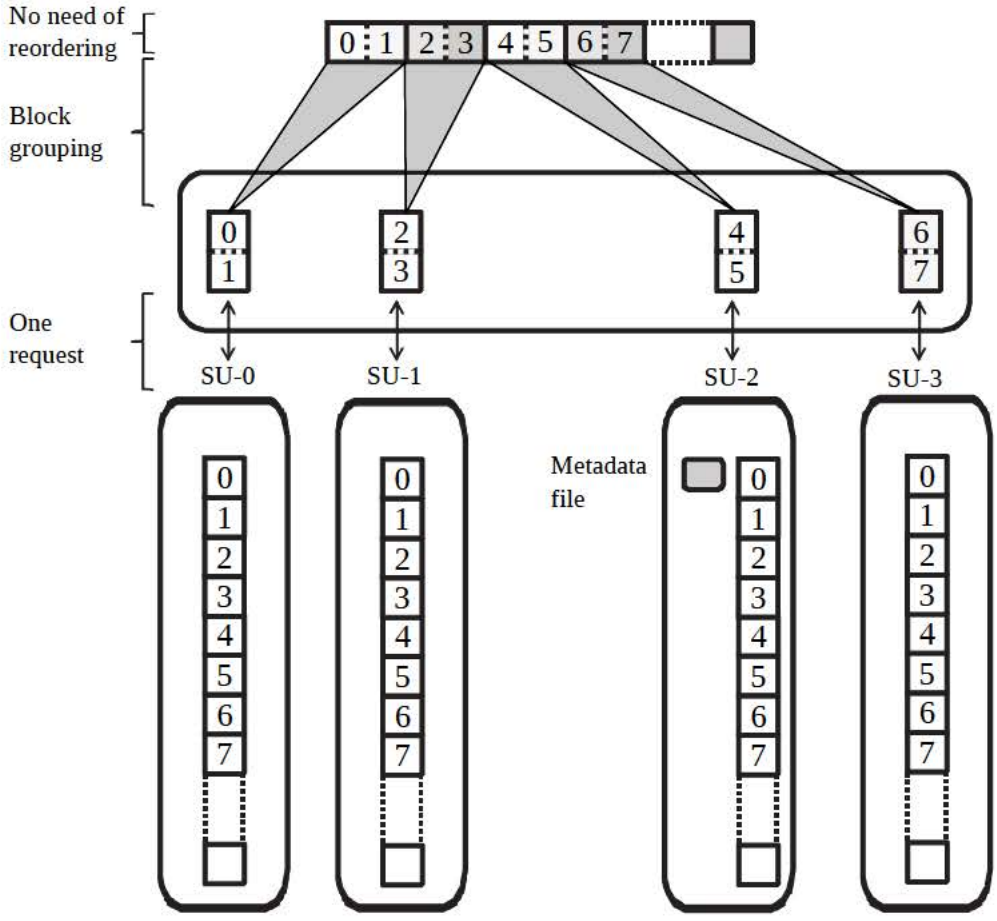


Figure 3.18: *Full replication with block grouping and no reordering*

3.5 Summary

This chapter has explained the architecture and implementation of a generic I/O middleware specifically suited for large-scale distributed systems. To implement this architecture we have made used of *Expand*, the parallel file system for clusters and supercomputers described in chapter A, as the base file system for this proposal. In order to use *Expand* as a generic file system for large-scale distributed systems it needed several modifications and enhancements. This chapter has detailed these modifications and enhancements made to *Expand* to adapt it to large-scale distributed systems.

Chapter 4

Evaluation of the parallel file system for large-scale distributed systems in Grid and volunteer computing environments

This chapter presents the main results of the evaluation of the generic I/O middleware proposed in this thesis: a parallel file system for data access in large-scale distributed systems, and its application in Grid and volunteer computing environments.

4.1 Introduction

The two different target environments will be analyzed independently. First of all, the results of the evaluation for grid environments will be shown. Later, its application for volunteer computing environments.

To do these evaluations, a complete simulator of the proposed system has been implemented; so that, the evaluations can be done in a more flexible way, and without the requirement of having real hardware environments to do the tests.

4.2 Evaluation environments

This section explains the different evaluation environments used to do the tests. There are three real platforms. Also, a simulator has been developed to extend the tests to cases that were not feasible to perform in real environments.

4.2.1 Real environments

In order to analyze the performance of the proposed architecture under networks with different latencies, four hardware platforms have been used for the evaluations:

Cluster (ARCOS cluster) The ARCOS platform is a cluster of 24 nodes, Intel® Xeon® CPU E5405 at 2.00 GHz, 4 cores/CPU, with 4 GB of RAM, connected through a Gigabit Ethernet, running Ubuntu Linux 64-bit with kernel 2.6.35-32-server. This cluster has a low latency level, because experiments conducted in this cluster have been done in isolation.

Desktop Grid (UC3M computer labs) The UC3M computer labs platform are five computer labs of Universidad Carlos III de Madrid, spread across two buildings, used as laboratories by teachers and students. The five computer labs sums up to 95 workstations AMD Athlon™ 64 X2 Dual Core Processor 4200+, with 2 GB RAM. This platform has a medium latency level, because experiments conducted in this cluster have been done sharing the network with other users in the university.

Small Grid (GRIDIMadrid) GRIDIMadrid is a Grid environment for research purposes in Comunidad de Madrid. It is formed of heterogeneous servers located in different places separated by a distance ranging from 0 to 50 km (see Figure 4.1). The servers belong to different 13 research institutions and universities of Madrid. All the machines used run The Globus Toolkit version 4. This platform has a medium-high latency level, because experiments conducted in this platform have been done using machines located in different cities around Madrid, and sharing the network with other users.

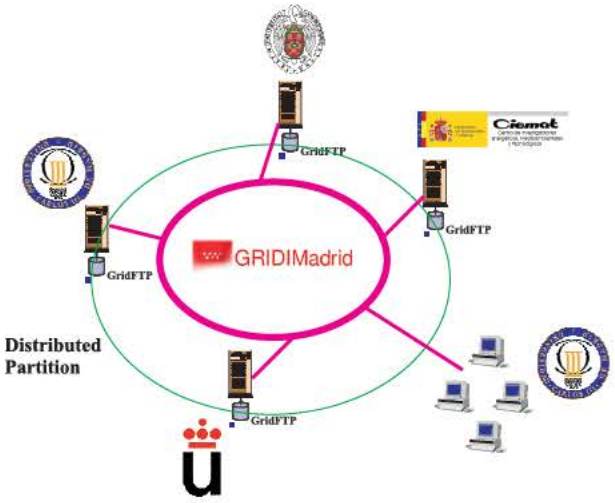


Figure 4.1: *GRIDIMadrid platform*

Medium Grid (Grid5000) The Grid5000 platform is an heterogeneous Grid of 1195 nodes, 2266 processors, 8184 cores, each node connected through a Gigabit Ethernet, running Debian Linux 64-bit with kernel 2.6.32-5-amd64. The nodes are organized in 25 clusters, in 10 cities of France, with a 10 Gigabit Ethernet backbone. Figure 4.2 shows the interconnection network of Grid5000 cities. This platform has a high latency level, because experiments

conducted in this platform have been done using machines located in different cities around France, and sharing the network with many other users.

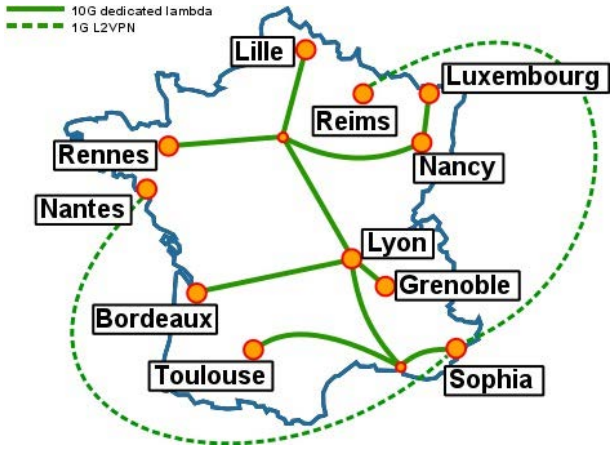


Figure 4.2: *Grid5000 interconnection schema*

The software libraries used for the evaluations are:

Grid software Globus Toolkit 5.2.2.

HTTP software Apache web server 2.0.

4.3 Evaluation in Grid environments

This section details the study performed of the general architecture for data access in grid environments, and the benchmarks used for that purpose.

4.3.1 Objective

The main objective of the evaluation is to compare two different models of accessing the data:

Classic model The classic model is that in which the files are first downloaded, then they are written to disk, and finally, read (see Figure 4.3(A)).

Remote access model In the remote access model the data are accessed on-the-fly, downloading the data blocks when needed, and never written to disk (see Figure 4.3(B)).

The specific objectives of the evaluation are:

- To study the effect of reading part of a file: 1%, 5%, 10%, 20%, ... 100%
- To study the effect of different protocols used in distributed environments, namely, GridFTP, XIO (with GridFTP driver), and HTTP

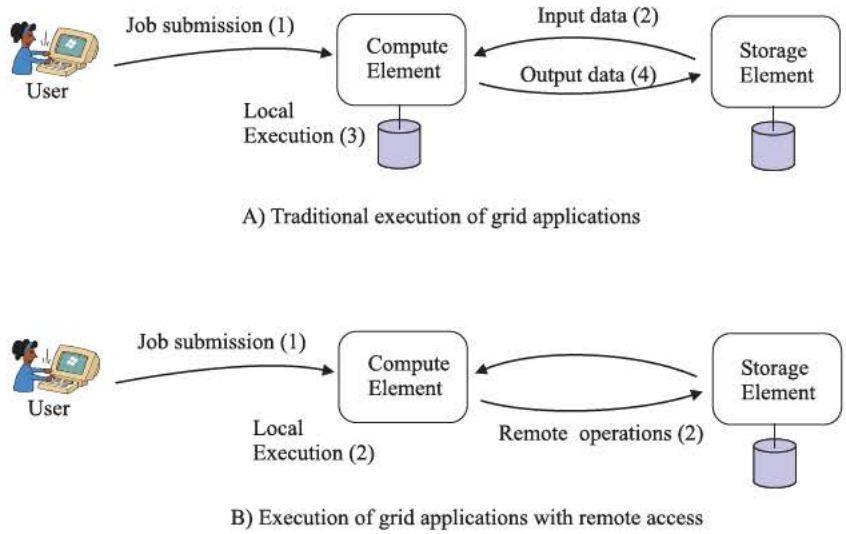


Figure 4.3: *Classic vs Remote models*

- To study the effect of having balanced/unbalanced workloads on servers: 0%, 10%, 20%, ..., 100%
- To study the effect of having different client workloads: Subset, Random, Proportional, Parallel, and Expand
- To study the effect of having networks with different latencies: low, medium, and high
- To study the effect of third-party transfers

4.3.2 Benchmarks definition

To evaluate the behavior of the general architecture for data access in Grid environments, several evaluations in different scenarios using I/O benchmarks have been used. Two different benchmarks have been designed for this thesis:

- Random benchmark
- Balanced benchmark

These tests are artificial benchmarks consisting of reading a set of files, or part of them, using different protocols.

4.3.2.1 Random benchmark

This benchmark evaluates remote access with grid services in a typical grid computing environment, in which users download a number of files for later processing. They download a set of files, or schedule a batch job for downloading the required files, and once the files are downloaded and written on the hard disk, they are processed. In this test, there is a fixed workload of 50 files of 100 MB each, which are assigned randomly to clients. Every client downloads a random number

of files, being 50 the total amount of files downloaded by all clients. This benchmark compares the classic method of download the files, write them to disk, and later, read them from disk ("Classic" in figures); versus remote access to data, processing the files on-the-fly instead ("Expand" in figures), using an internal buffer size of 10% of file size, 10 MB. The benchmark also evaluates the impact of reading only part of the file, instead of the whole file.

Three different download scenarios are evaluated:

One server All files are stored in one server, and clients download the files from this server.

Distributed copies The files are distributed among a number of servers, and clients download each file from the corresponding server.

Parallel All files are replicated in all the servers, and clients download the files in parallel from all the servers.

4.3.2.2 Balanced benchmark

This test is an extension of the above, Random benchmark. In this case, instead of assigning the workload to clients randomly, the workload is forced to produce a certain balance (or unbalance) in the servers (the workload that the servers must serve), so that, some percentage of the workload is balanced among all the servers, and the rest is assigned to one server. The different scenarios can be shown in Figure 4.4:

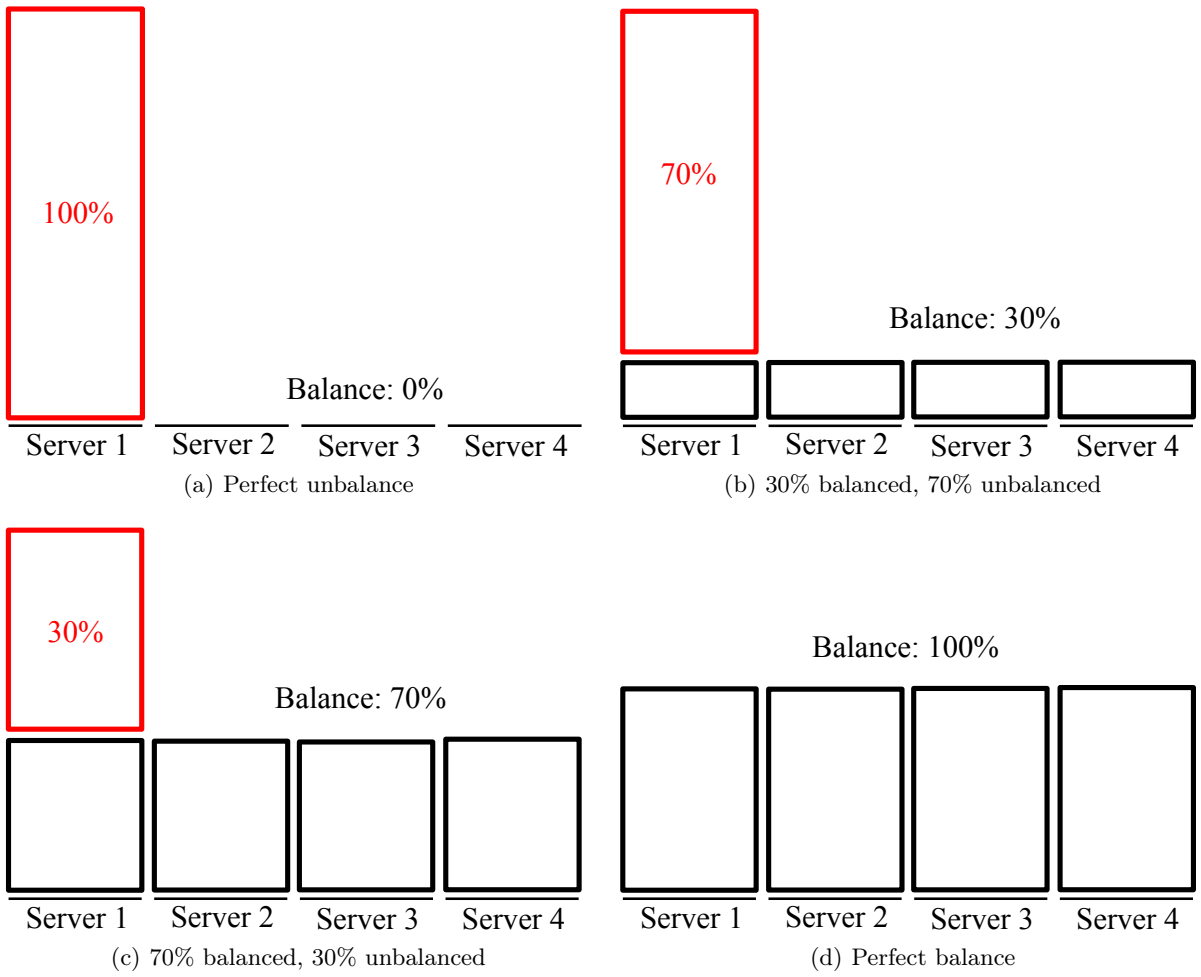
Perfect unbalance This can be considered the worst situation. In this case, all the workload is served by only one server (Figure 4.4(a)).

Certain balance/unbalance In this case the workload has some degree of balance (or unbalance). For example, if there is 30% of balance (Figure 4.4(b)), this means that 30% of the workload is served evenly by all the servers, and the remaining 70% is served by only one server. Alternatively, if there is 70% of balance (Figure 4.4(c)), this means that 70% of the workload is served evenly by all the servers, and the remaining 30% is served by only one server.

Perfect balance This can be considered the best situation. In this case, all the workload is served evenly by all the servers (Figure 4.4(d)), so that, all the servers serve exactly the same number of files.

Once a certain balance level on the servers is chosen, we need a client workload that generates the chosen balance level in the servers. To force some level of workload balance in the servers, several ways of distributing that workload among clients can be used. What follows is a list of different ways of generating client workloads, so that, each of them is a different way of producing the chosen balance level in the servers:

Subset In this case, every client has an associated server, so that, clients request every file to their associated server (see Figure 4.5(a)). This case is similar to Distributed copies (from Random benchmark), the files needed by a client are located and, thus, served by just one server, but in this case, all the files needed by a certain client are always downloaded from the same server.



(a) Perfect unbalance

(b) 30% balanced, 70% unbalanced

(c) 70% balanced, 30% unbalanced

(d) Perfect balance

Figure 4.4: *Balance levels*

Random In this case, each time a client needs to access a file, it selects a server randomly among the available servers (see Figure 4.5(b)).

Proportional In this case, every time a client needs to access a file, it selects a different server, so that, client's workload is proportional among servers (see Figure 4.5(c)).

Parallel In this case, every file is downloaded in parallel, evenly from all available servers (see Figure 4.5(d)). Thus, this case is perfectly balanced (100%), i.e., it generates equally balanced workload on servers. For this reason, this case will only appear at 100% balance level in the figures.

Expand This case is very similar to the parallel case (see Figure 4.5(d)), but accessing files remotely, instead of downloading and writing them to disk before reading. For the same reason as in parallel, this case generates perfectly balanced workload on servers, and, therefore, this case will only appear at 100% balance level in the figures.

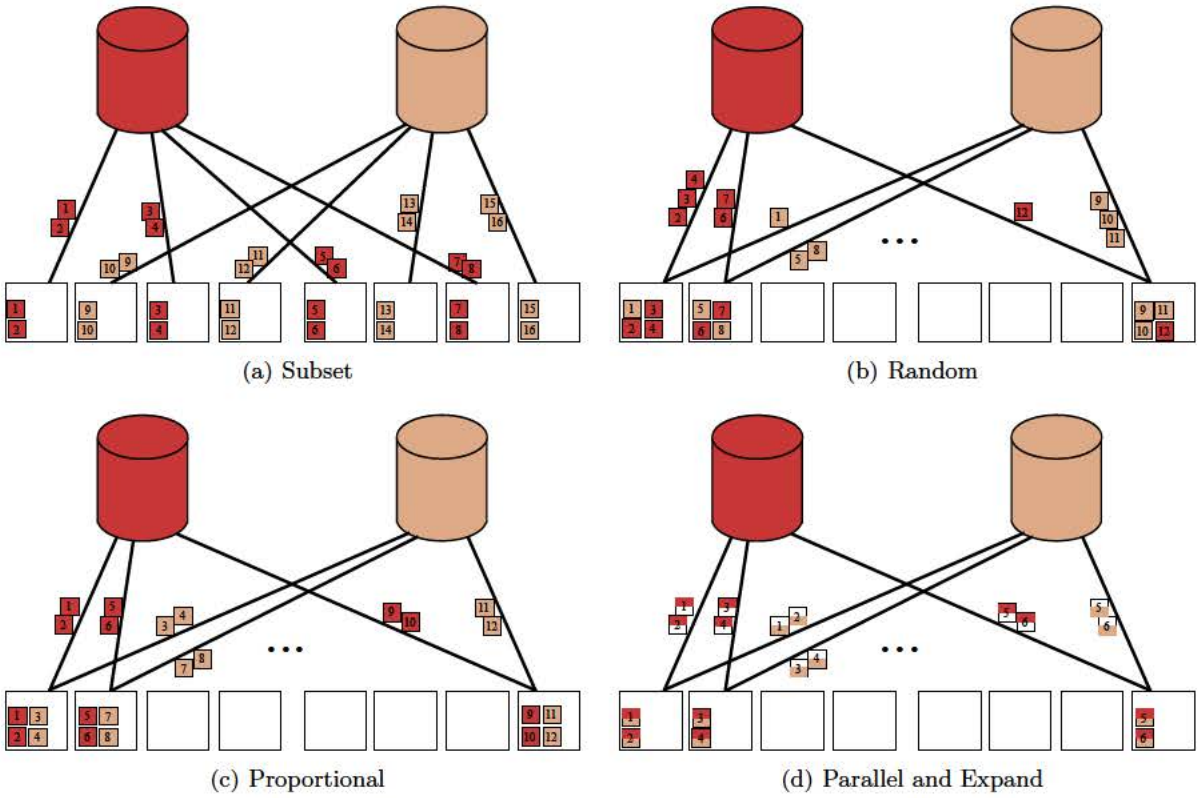


Figure 4.5: *Balance modes*

As in Random benchmark, the workload consists of 50 files of 100 MB for the experiments conducted in ARCOS cluster, and 128 files of 100 MB for the experiments conducted in Grid5000. The remote access model uses an internal buffer size of 10% of file size, 10 MB.

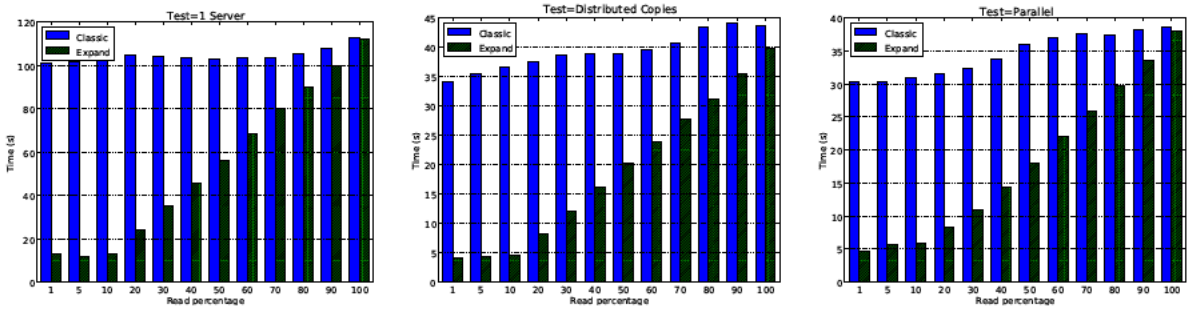
4.3.3 Results, analysis, and discussion of evaluation in real environments

The specific objectives of these tests are:

- To study the effect of reading part of a file: 1%, 5%, 10%, 20%, ... 100%
- To study the effect of different protocols used in distributed environments, namely, GridFTP, XIO (with GridFTP driver), and HTTP
- To study the effect of having balanced/unbalanced workloads on servers: 0%, 10%, 20%, ..., 100%
- To study the effect of having different client workloads: Subset, Random, Proportional, Parallel, and Expand
- To study the effect of having networks with different latencies: low, medium, and high
- To study the effect of third-party transfers

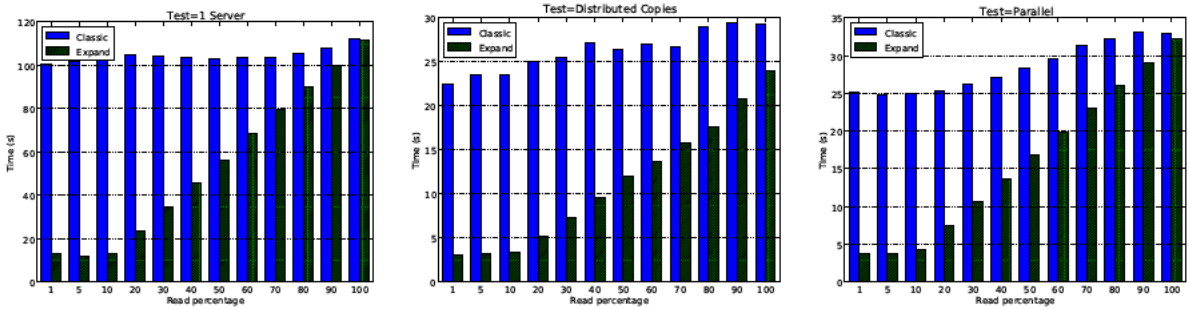
4.3.3.1 Effect of reading part of a file

Though obvious, it is worth noting how beneficial remote access is. Figures 4.6 and 4.7 capture the effect of reading part of a file in Random benchmark for a number of servers. Compared to the classic approach which downloads a file completely to disk before reading, remote access outperforms significantly.



(a) 1 Server (b) Distributed Copies (c) Parallel

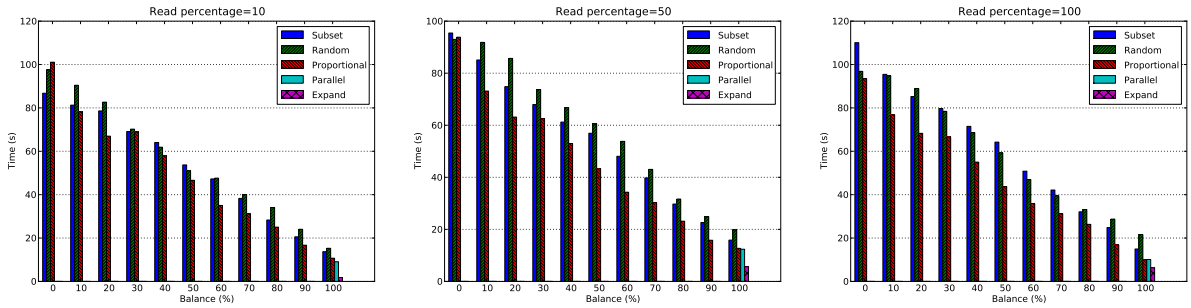
Figure 4.6: *Effect of reading part of a file in Random Benchmark (HTTP, 16 clients, 4 servers, Cluster platform)*



(a) 1 Server (b) Distributed Copies (c) Parallel

Figure 4.7: *Effect of reading part of a file in Random Benchmark (HTTP, 16 clients, 8 servers, Cluster platform)*

Figure 4.8 captures the effect of reading part of a file in Balanced benchmark. Again, compared to the rest of approaches remote access always outperforms by a substantial difference.



(a) 10% (b) 50% (c) 100%

Figure 4.8: *Effect of reading part of a file in Balanced Benchmark (HTTP, 16 clients, 8 servers, Cluster platform)*

4.3.3.2 Effect of different protocols used in distributed environments

Four different protocols have been used in this thesis:

- OGSA ByteIO (Expand-WS and Expand-GridOGSA-ByteIO in figures)
- GridFTP
- Globus XIO (with GridFTP driver)
- HTTP

First of all, OGSA ByteIO protocol was evaluated to study the feasibility of this protocol for implementing a parallel file system. With this aim, the evaluation has been made using a single transfer and a typical grid computing scenario, both sending the data in plain text and encrypted [Bergua et al. (2008)].

To test the single transfer we have used the `globus-url-copy` command (which uses GridFTP protocol), *Expand* with GridFTP protocol and *Expand* with OGSA ByteIO protocol (Expand-WS in figure), sending the data in plain text and encrypted. Each job transfers a 500 MB file from client to server. The platform used was the Small Grid.

Figure 4.9 shows time, in seconds, required to run the single transfer benchmark. The best results are for the `globus-url-copy` command and *Expand* with GridFTP, which show similar results compared to OGSA ByteIO (Expand-WS in figure), but *Expand* being better than `globus-url-copy`, as expected.

For analyzing OGSA ByteIO in a typical grid scenario we have defined two grid benchmarks, one sends the data in plain text and the other encrypted, that consist of 500 jobs scheduled on 4 workstations. Each job accesses to a random number of files (between 1 and 10 files) chosen among 1000 files. The size of each file is 500 MB. This benchmark has been tested in different modes:

- All files are stored in one single GridFTP server (1 site in figures) and they are accessed using the `globus-url-copy` command, the command line tool provided by Globus.

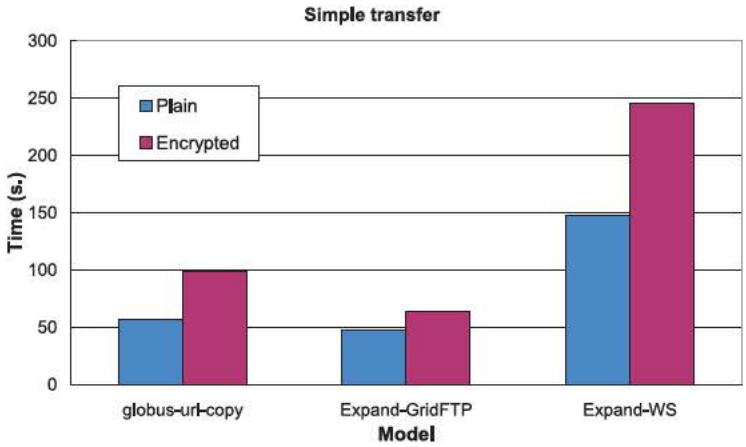


Figure 4.9: *Simple Transfer in Expand (Small Grid platform)*

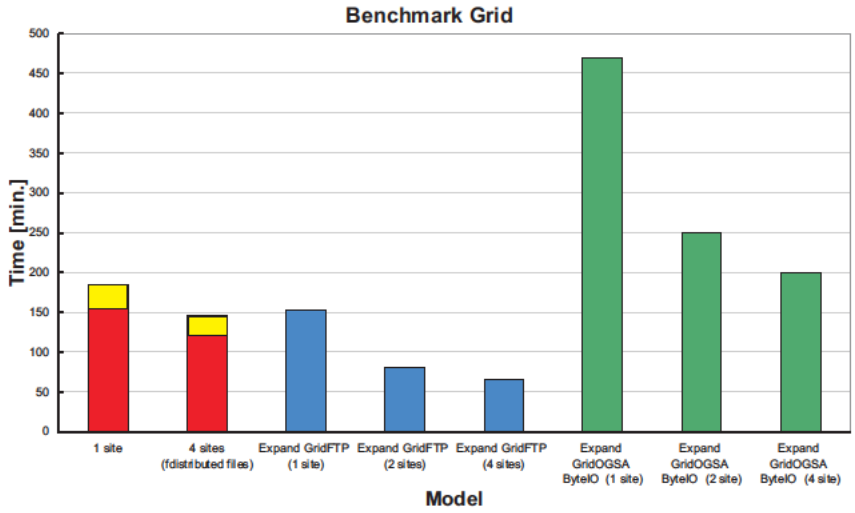
- All files are replicated in 4 GridFTP servers (4 sites in figures). Each server stores 1000 files, but in this case each file is accessed in parallel using the 4 servers.
- Using *Expand* with GridFTP protocol (Expand-GridFTP in figures) and several number of servers for the distributed partition (1, 2, and 4). The files are accessed using POSIX system calls.
- Using *Expand* with OGSA ByteIO protocol (Expand-GridOGSA-ByteIO in figures) and several number of servers for the distributed partition (1, 2, and 4). The files are accessed using POSIX system calls.

In the first two scenarios described below, the application transfers previously the whole file to the local node and then processes it. In the case of using *Expand* the application accesses the files directly and remotely, which avoids writing in the local node. To perform this test we have used 4 workstations as clients and 4 GridFTP servers in the Small Grid platform.

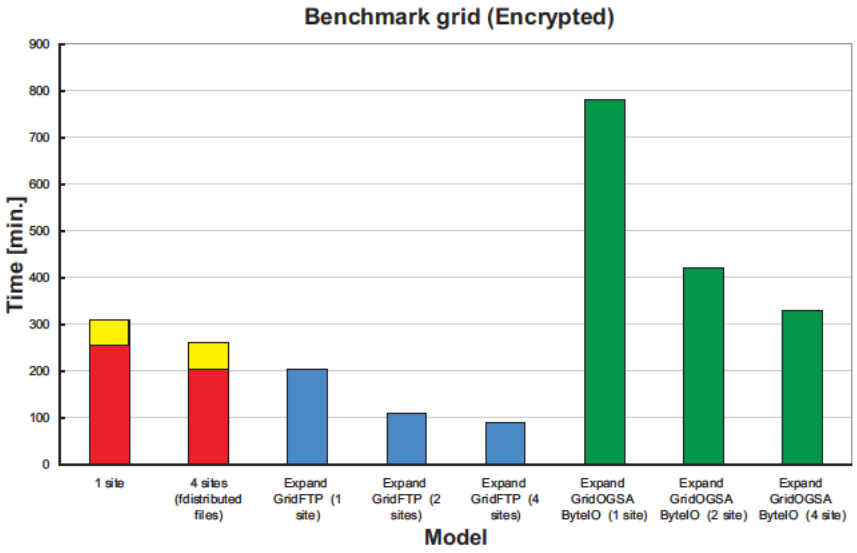
Figure 4.10 shows time, in minutes, required to run the benchmark. In the case of the scenarios who uses Globus (fist two bars), they show the time required to completely transfer the file to the node (dark part of the bar) and processing time (light part of the bar). The best results are obtained for *Expand* because it accesses the files directly and does not require writing the files previously to process them. The best results are for a partition of 4 servers in *Expand* because it accesses the files in parallel. This test probes that the parallel file system proposed in this thesis can be used as a generic I/O middleware in real Grid environments and that good results in accessing to data can be obtained.

As in the single transfer case, the results of OGSA ByteIO implementation were clearly the worst. Therefore, the OGSA ByteIO implementation was discarded in the rest of experiments due to its poor performance compared to GridFTP [Bergua et al. (2008)].

In grid environments, the GridFTP protocol is a standard. Globus XIO is a framework that offers a simple interface to the underlying protocols in the Grid. In this thesis an implementation of Globus XIO with a GridFTP driver has been chosen. In other distributed systems HTTP is also a well-known standard.



(a) Plain

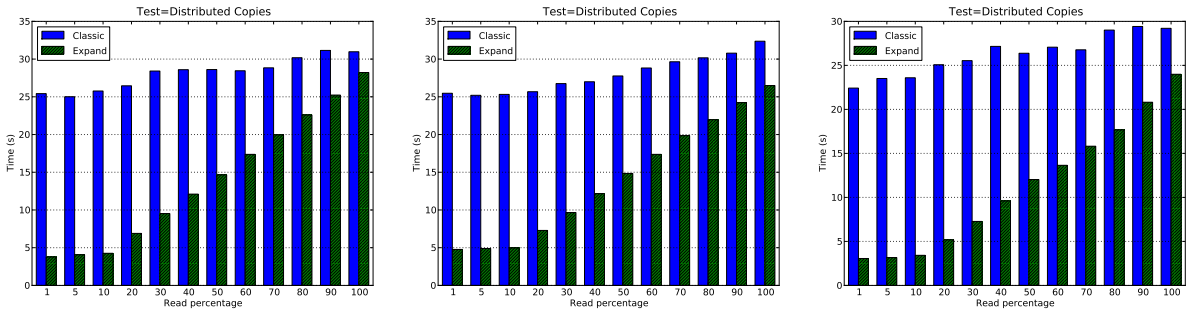


(b) Encrypted

Figure 4.10: *Benchmark Grid of Expand (Small Grid platform)*

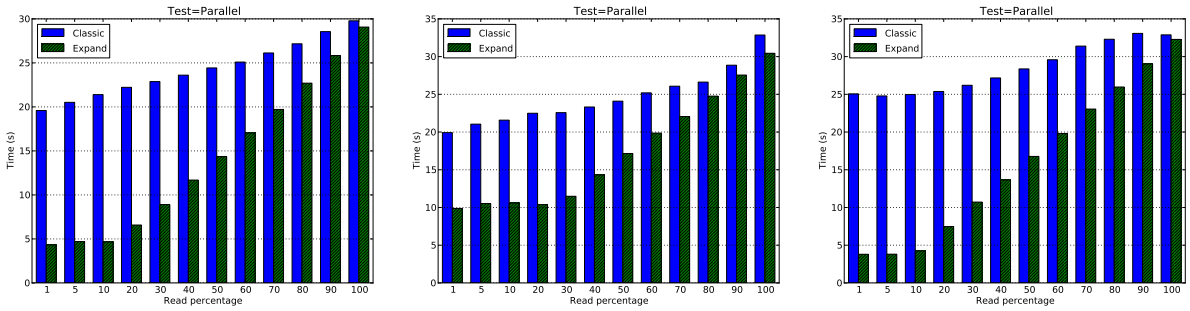
Figures 4.11 and 4.12 show the effect of the different protocols used in distributed environments using Random benchmark with Distributed copies and Parallel modes.

Similar conclusions can be drawn from figures 4.11 and 4.12. GridFTP behaves very similar to HTTP, although the latter performs a little bit better, as expected since HTTP is much simpler than GridFTP. However, Globus XIO seems to behave worse than GridFTP in remote access mode ("Expand" in figures). Although the Globus XIO implementation for this thesis used a GridFTP driver underneath, Globus XIO is a framework composed of stacked drivers (for the purpose of easing the development of communication protocols), causing an extra overhead compared to using



(a) GridFTP (b) XIO (c) HTTP

Figure 4.11: *Effect of different protocols used in distributed environments using Random Benchmark with Distributed Copies mode (16 clients, 8 servers, Cluster platform)*

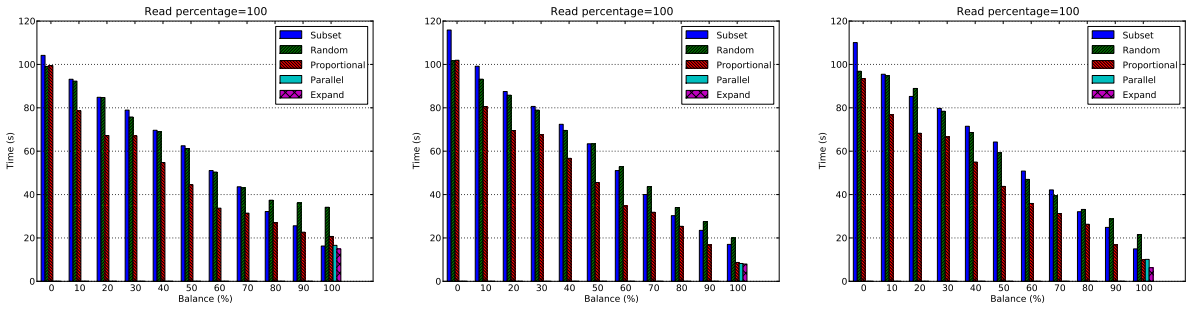


(a) GridFTP (b) XIO (c) HTTP

Figure 4.12: *Effect of different protocols used in distributed environments using Random Benchmark with Parallel mode (16 clients, 8 servers, Cluster platform)*

GridFTP directly when doing remote access.

Figure 4.13 shows the effect of the different protocols in Balanced benchmark used in distributed environments.



(a) GridFTP (b) XIO (c) HTTP

Figure 4.13: *Effect of different protocols used in distributed environments in Balanced Benchmark (16 clients, 8 servers, Cluster platform)*

As showed, the protocols showed very similar behaviors. This means that the protocols used in the experiments do not introduce any significant bias in the results.

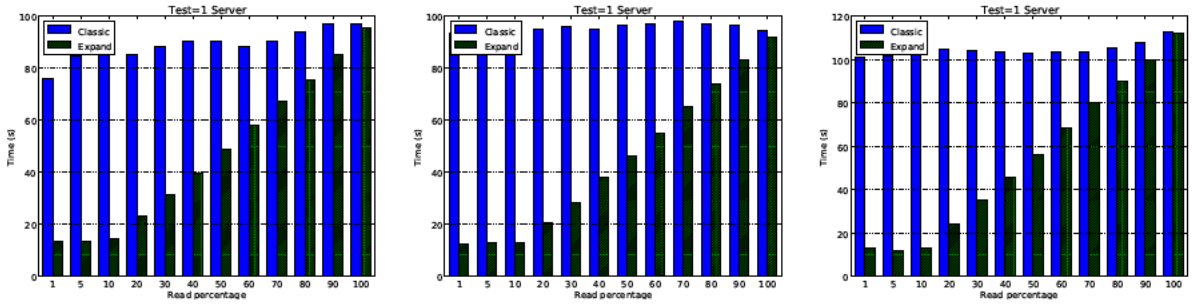
4.3.3.3 Effect of different workloads on servers and clients

This section will analyze the effect of different workloads on servers and clients. As described in section Benchmarks definition, two artificial benchmarks have been used:

- Random benchmark
- Balanced benchmark

Random benchmark As described in section 4.3.2.1, Random benchmark evaluates remote access with grid services in a typical grid computing environment. It compares three modes of distributing files across a set of servers: all files stored in one server, distributed copies, and parallel access. The first conclusion that can be drawn is that any mode that use more than one server is better that "1 Server" mode, as expected. Also, "Parallel" outperforms "Distributed copies" in "Classic" modes (see Figures 4.6 and 4.7).

Going deeper, if there is only one server (see Figure 4.14), remote access ("Expand" in figures) clearly outperforms "Classic" mode when reading only some part of the files. However, when read percentage reaches 100% remote access does not beat "Classic" mode so clearly, because the server is so overwhelmed that becomes the limiting factor. For the same reason, increasing the number of clients but with a fixed total workload, results are equivalent because total execution time is limited by server throughput.

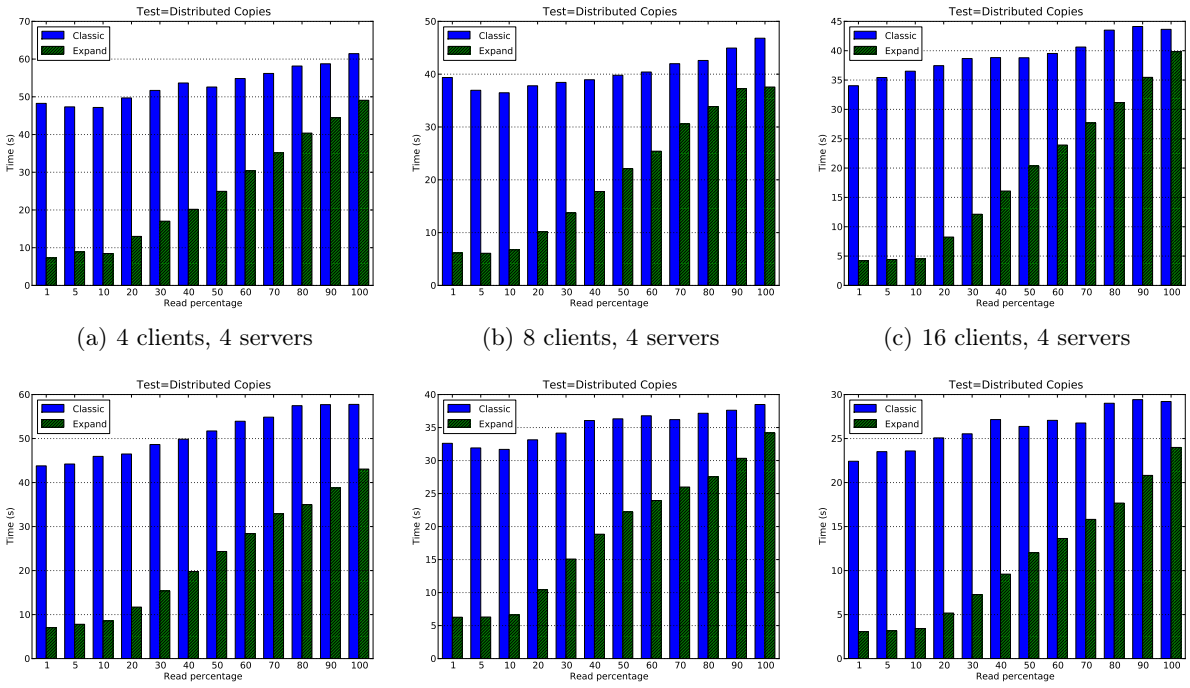


(a) 4 clients (b) 8 clients (c) 16 clients

Figure 4.14: *Random Benchmark results for 1 server (HTTP, Cluster platform)*

In the "Distributed copies" scenario (see Figure 4.15), increasing the number of servers reduces total execution time, as expected. And, increasing the number of clients also reduces total execution time, since there are more clients for the same workload, so every client finishes sooner. In every situation, remote access outperforms traditional mode.

In the "Parallel" scenario (see Figure 4.16), again, increasing the number of servers reduces total execution time, as expected. However, increasing the number of clients does not necessarily reduce the execution time, since there are more clients generating network requests, which means more work for the servers. It seems to reach the shortest execution time when using 8 clients. In every situation, remote access outperforms traditional mode.



(a) 4 clients, 4 servers (b) 8 clients, 4 servers (c) 16 clients, 4 servers

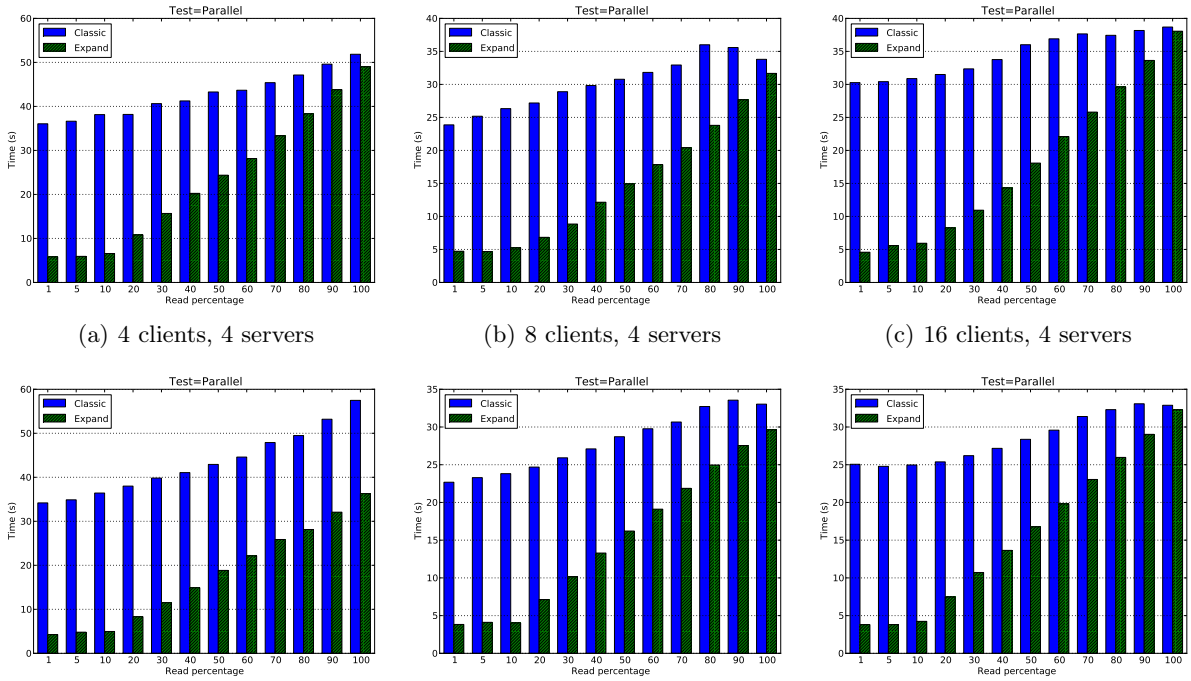
(d) 4 clients, 8 servers (e) 8 clients, 8 servers (f) 16 clients, 8 servers

Figure 4.15: *Random Benchmark results with Distributed Copies mode (HTTP, Cluster platform)*

Balanced benchmark As described in section 4.3.2.2, Balanced benchmark evaluates a number of levels of balance (or unbalance) on servers: 0% (Perfect unbalance), 10%, 20%, ..., 100% (Perfect balance). And, for every server's balance level, five different client workloads were tested: Subset, Random, Proportional, Parallel, and Expand. As explained in section Balanced benchmark, Parallel and Expand cases are only tested in 100% balance level (Perfect balance).

Figure 4.17 shows a combination of number of clients and servers for 100% read percentage. When using 4 clients, "Subset" mode is clearly worse than others (see Figures 4.17(a) and 4.17(b)). When using 16 clients "Subset" is comparable to others (see Figures 4.17(c) and 4.17(d)). Generally, "Proportional" provides very good performance for any balance level.

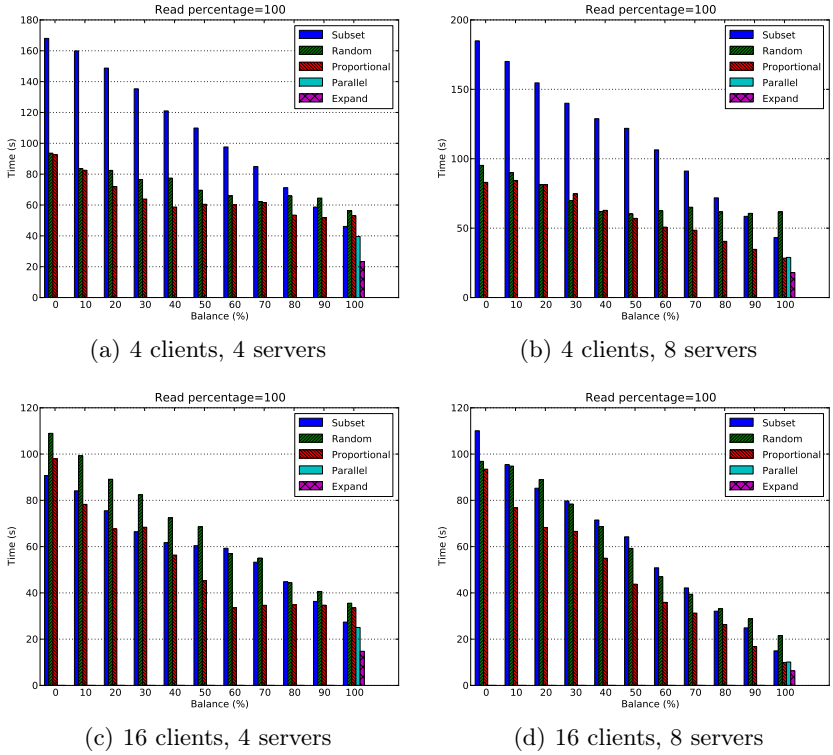
However, neither server balance nor client workloads are things that can be chosen in a real scenario. They simply happen. The combination of "Subset", "Random", and "Proportional" modes with server balance levels ranging from 0% to 100% pursues the aim to compare "Parallel" and "Expand" with all the possible scenarios that might take place in real life. The results show that parallel access outperforms significantly any other scenario, and remote access ("Expand" in figures) also outperforms parallel access (see Figure 4.17).



(a) 4 clients, 4 servers (b) 8 clients, 4 servers (c) 16 clients, 4 servers

(d) 4 clients, 8 servers (e) 8 clients, 8 servers (f) 16 clients, 8 servers

Figure 4.16: *Random Benchmark results with Parallel mode (HTTP, Cluster platform)*



(a) 4 clients, 4 servers (b) 4 clients, 8 servers

(c) 16 clients, 4 servers (d) 16 clients, 8 servers

Figure 4.17: *Effect of having balanced/unbalanced workloads on servers (HTTP, Cluster platform)*

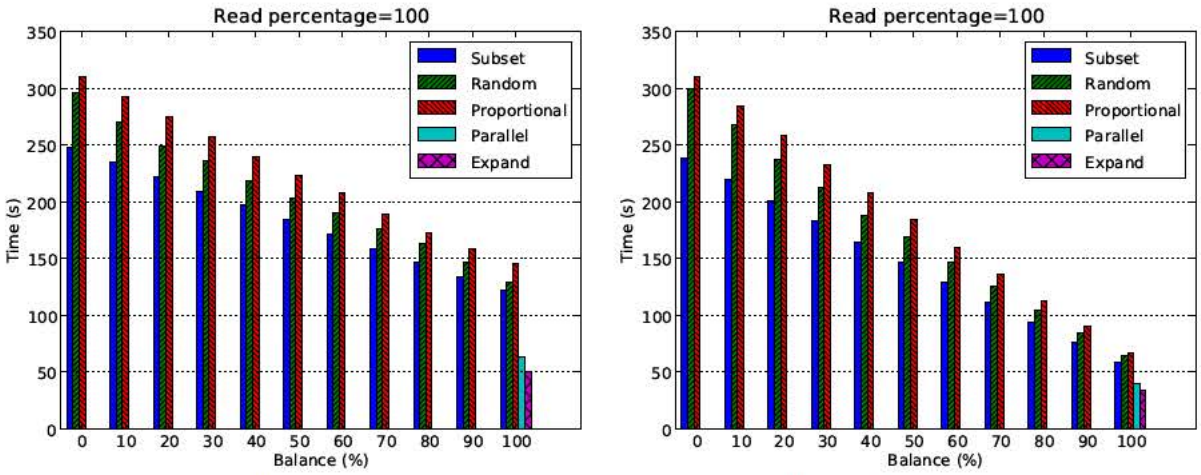
4.3.3.4 Effect of having networks with different latencies

Earlier tests were carried out in an isolated cluster. This controlled environment used for the experiments was the Cluster platform. Experiments conducted in this platform were carried out in isolation, i.e., no other experiments or users were using the cluster.

However, to study the effect of having networks with different latencies non-controlled platforms are necessary. To this extent, the Medium Grid platform has been used. This section will show the performance of the Balanced benchmark under different configuration of clients and servers in the Medium Grid platform.

Lyon multi-cluster Lyon is one the cities that shares more nodes to Grid5000 by means of four clusters with a total sum of more than 100 nodes. This city allows to conduct experiments in a multi-cluster environment while other users are using the platform. In this experiment, 16 clients ran all in Lyon, while the servers were distributed across the rest of Grid5000.

Figure 4.18 shows the results carried out.



(a) 16 clients, 2 servers

(b) 16 clients, 4 servers

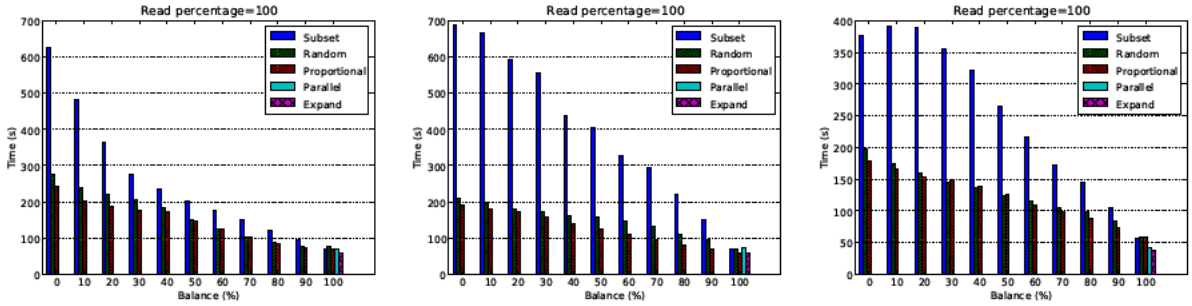
Figure 4.18: *Effect of having networks with different latencies (HTTP, all clients at Lyon, Medium Grid platform)*

The results show that both "Parallel" and "Expand" clearly outperforms traditional scenarios, but remote access offers a significant advantage over plain parallel access.

Grid5000 In order to test remote access in an even more realistic scenario, we have used a distributed configuration of clients and servers around Grid5000.

Next figures show results using the whole Grid5000. In particular, Figure 4.19 shows results in Grid5000 while Figure 4.20 uses Grid5000 without Lyon (the city that contributes with more nodes).

Results show now that while "Parallel" mode is comparable to traditional scenarios, but not always is better, "Expand" becomes the best alternative.



(a) 8 clients, 8 servers (b) 8 clients, 16 servers (c) 16 clients, 8 servers

Figure 4.19: *Effect of having networks with different latencies (HTTP, Medium Grid platform)*

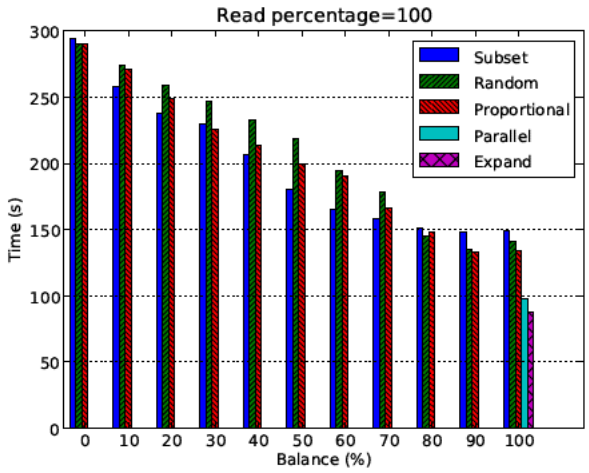
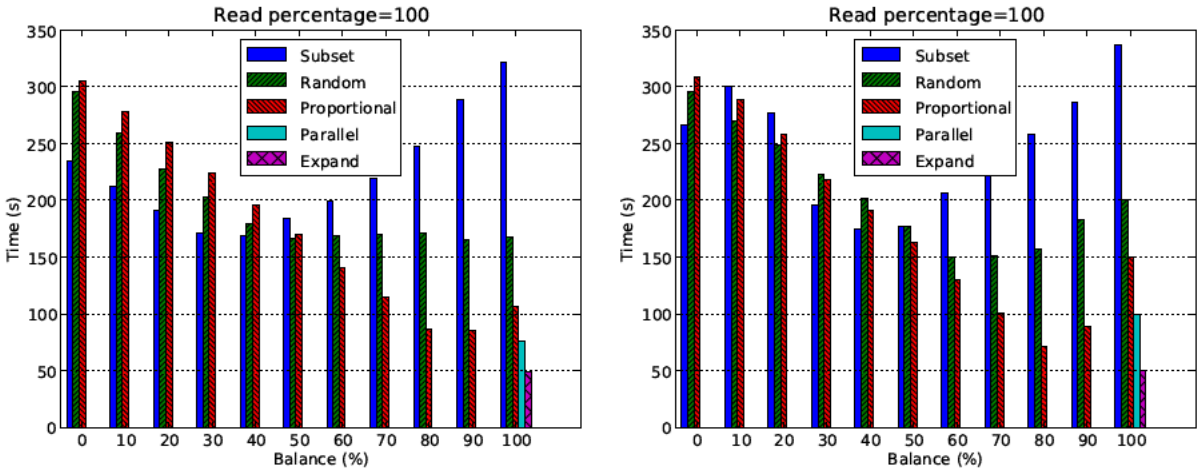


Figure 4.20: *Effect of having networks with different latencies (HTTP, 16 clients, 4 servers, Medium Grid platform without Lyon)*

Presence of slow links In previous experiments conducted in Grid5000 all the cities used for the experiments were connected through 10G links. This section will explore the situation in which a slow link is used. In particular, Reims, which has a 1G link (see Figure 4.2), has been used.

The presence of a slow link creates a turning point for "Subset" and "Random" modes when reaching a balance level ranging 40%-60%, and around 80% for "Proportional". From that point on, times get bigger and bigger. This is so because while the balance level keeps low, faster servers, that serve most of the data, hide the penalty of the slow link. But from that point on, the load of the slow server is so high that the clients must wait for that server to serve the data (see Figures 4.21 and 4.22).

While parallel access offers much better performance than traditional modes, "Expand" becomes again as a superior choice.



(a) 16 clients, 8 servers

(b) 16 clients, 16 servers

Figure 4.21: *Effect of having networks with different latencies in the presence of slow links (HTTP, all clients at Lyon, Medium Grid platform)*

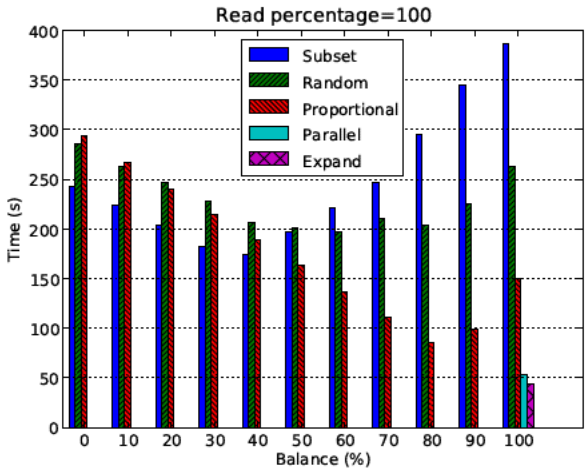


Figure 4.22: *Effect of having networks with different latencies in the presence of slow links (HTTP, 16 clients, 8 servers, Medium Grid platform without Lyon)*

4.3.3.5 Effect of third-party transfers

Another common scenario of file transfers by grid applications when source data are stored on one single server or data repository is that the grid application orders the transfer on the input data from the data server to the parallel file system used inside the cluster. The data must travel from the data server to a node of the cluster and, from there, to the parallel partition. Then the application processes the input files to produce the results. The whole process can be shown in Figure 4.23 [Bergua et al. (2009b)].

In the first step the application requests a data set to a *downloader* which downloads the required data from the data repository through the internet using GridFTP or HTTP/HTTPS. The data is then downloaded from the data repository to the local node which the *downloader* is

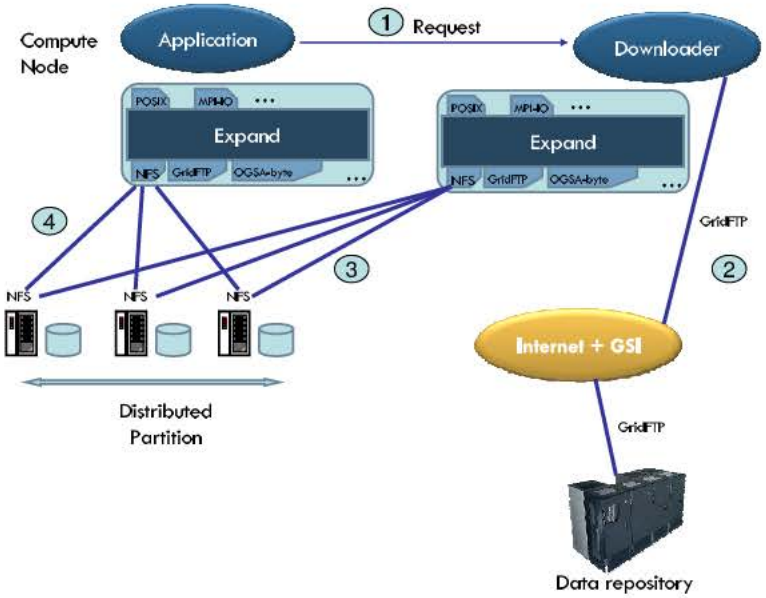


Figure 4.23: *Traditional model of third-party file transfer in Grid*

executing in and transferred to the parallel partition inside the cluster. Finally, the application can access to the data which resides in the parallel partition in parallel. For example, the StoRM SRM operates in a similar way.

However, there is still a bottleneck in the step number 2 labeled "GridFTP" in Figure 4.23. The main problem of this model is that the data must be serialized when downloaded from the data repository until they finally are copied to the parallel partition due to the use of intermediate nodes. In this case, the node the *downloader* runs in is the bottleneck because the data must come serially to the *downloader*. Only when the data are in the *downloader* node they are parallelized to the parallel file system in use, for instance, GPFS or "Expand+NFS" in Figure 4.23.

The new model that uses the parallel file system proposed in this thesis for doing third-party transfers ("Expand" in Figure 4.24), proposes that an *uploader* transfers the input data needed by the application from the data server directly to the parallel file system used by cluster, using the parallel file system proposed in this thesis, without going through any intermediate node in the cluster. After that, the application can access in parallel and remotely to the data with, possibly, a different protocol, in our case the NFS protocol is used to access the data locally inside the cluster.

There is a parallel file system's partition managed by *Expand* that can be accessed using two different protocols: GridFTP or HTTP from remote sites and NFS from local nodes. When the *uploader* transfers the data to *Expand* it would really access to a remote partition using GridFTP. When the data were in that partition the application would access locally to the data in this partition using the NFS protocol inside the cluster.

An *Expand* parallel partition which resides in a set of nodes can be accessed both remotely and locally. If accessed remotely GridFTP or HTTP is used and if accessed locally NFS is used, due to *Expand* can be used with different protocols. This characteristic allows an *uploader*, located remotely, to access directly to this partition without going through intermediate nodes.

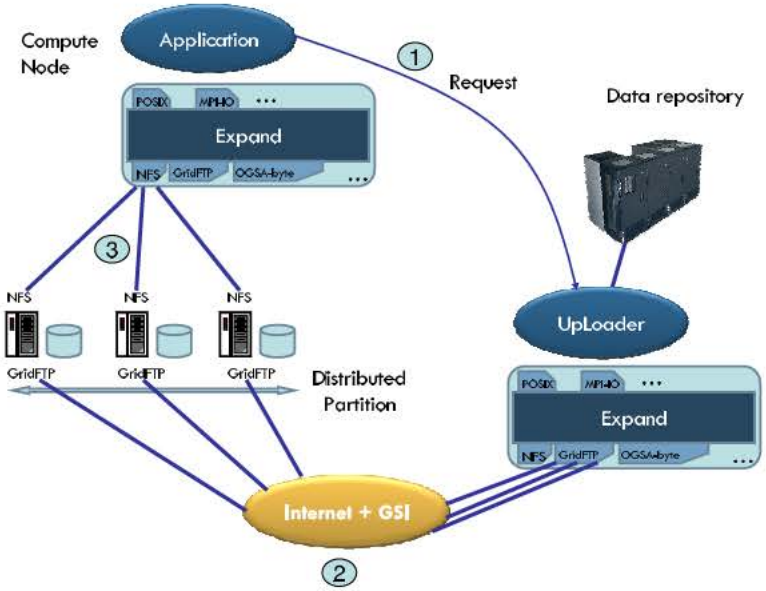


Figure 4.24: *New model of third-party file transfers in Grid with Expand*

Therefore, the proposed solution is based on remotely dumping the data over a parallel partition using *Expand* and GridFTP or HTTP. Once the data has been dumped the application would access the data using the same partition with the NFS protocol.

The use of this schema is different from the one provided by an SRM because an SRM exhibits a uniform interface to successfully interoperate as well as provide dynamic space allocation and file management on shared storage systems. They call on transport services to bring files into their space transparently and provide effective sharing of files.

The proposed solution uploads the data using *Expand*, which knows the details of the parallel partition, so it can transfer the data in parallel directly from the source. This cannot be done with GPFS o PVFS because when transferring data to those file systems the details of the parallel partition are not known, so the data must be transferred to an intermediate node which holds the parallel partition (typically mounted in the file system hierarchy), and which knows its details and how to write in parallel.

The main motivation for our performance test is to study whether the new proposed model of transferring data from the data server directly to the parallel file system is feasible, and if it achieves better performance. With this aim, the evaluation consists of multiple file transfers and different number of clients running in a cluster.

To test the multiple file transfer we have used *Expand* with NFS v3 protocol. In a job each client transfers 100 files of 2108160 bytes each (201.05 MB/client). We have launched 10 jobs and taken the mean of the results of each job.

For analyzing the system in a typical grid scenario we have defined two grid benchmarks. One copies the files from the data server to a local node and then it copies the files to the distributed partition. In the other, the files are copied from the data server directly to the distributed partition.

In both scenarios described above, after the input data files have been transferred to the parallel partition the application accesses each file directly and remotely, as is supposed to do

with a distributed file system.

To perform this test we have used the following platform:

- A data repository: Intel® Xeon® CPU Quad Core, 4 GB RAM
- 4 workstations as clients.
  - 2 of them: 4xIntel® Xeon® CPU Quad Core, 16 GB RAM.
  - 2 of them: 4xIntel® Xeon® CPU Dual Core Hyper-Threading, 16 GB RAM.
- 8 workstations as NFS v3 servers: each one Intel® Pentium® 4 CPU Hyper-Threading, 2 GB RAM.

All of them located in the campus of Leganés of Universidad Carlos III de Madrid.

Figure 4.25 shows bandwidth, in MB/s, obtained after running the multiple transfer benchmark 10 times and taking the mean. The best results are obtained using the new model proposed for 8 NFSv3 servers and 32 clients. The benchmark proves that the new model proposed (lines labeled 3 and 4 in figure 4.25) achieves better performance than the traditional one (lines 1 and 2, respectively) when using 32 clients [Bergua et al. (2009b)].

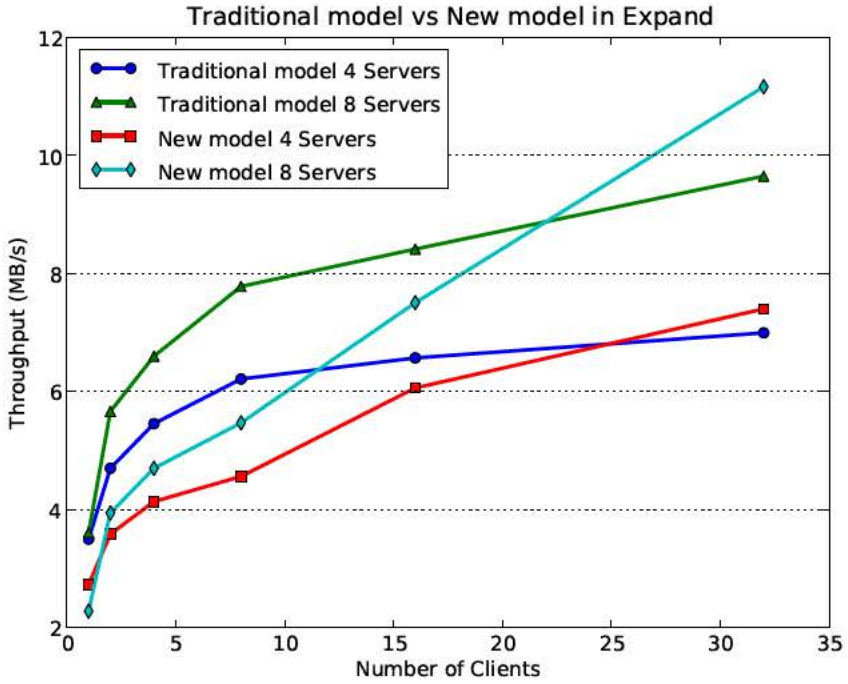


Figure 4.25: *Expand as third-party downloader evaluation results (two Cluster platforms)*

4.4 Evaluation in volunteer computing environments

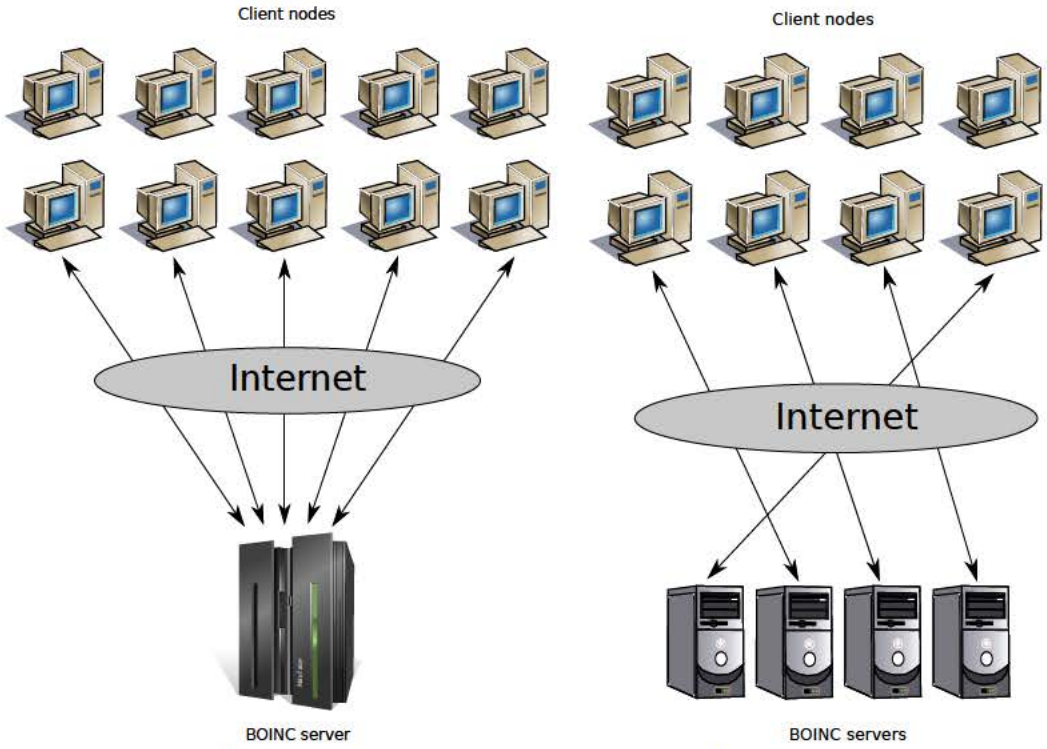
This section details the study performed of the general architecture for data access in volunteer computing environments, and the benchmarks used for that purpose.

4.4.1 Introduction

As can be shown in Figure 4.26(a), the central BOINC server might be a bottleneck on a BOINC project. This is a real problem when the number of clients increase and one of the main reasons is the high data traffic between clients and the data server.

As [Anderson (2004), Costa et al. (2008)] suggest, in the current implementation of BOINC, data distribution is achieved through the use of multiple centralized HTTP servers that share data with the entire network (see Figure 4.26(b)).

However, this proposal lack of the benefits of parallel access to the files because it downloads and uploads the input/output data files sequentially.



(a) Strict Master/worker model (b) Distributed data servers

Figure 4.26: *Strict Master/worker and distributed models in BOINC*

A new model of transferring data in volunteer computing environments is proposed. As shown in Figures 4.26(a) and 4.26(b) traditional volunteer applications transfer its input data from a data server to the local file system used in the worker node through the internet, then the application processes those files to produce the results, and, finally, the output files are transferred to the project data server. We propose a novel model, as shown in Figure 4.27, based on a generic

I/O middleware for large-scale distributed systems that transfers the input data needed by the application, and the output data produced by the application, from/to a pool of data servers in parallel using the parallel file system proposed in this thesis.

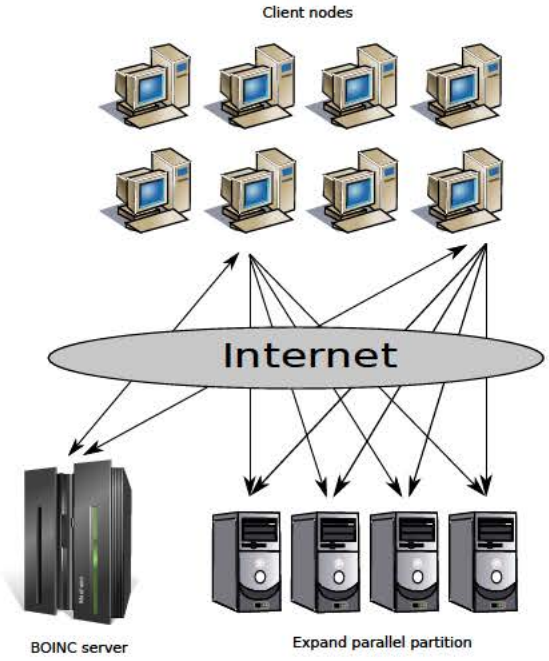


Figure 4.27: *Proposed model using Expand*

4.4.2 Integration of a parallel file system into BOINC

In order to evaluate the generic I/O middleware for large-scale distributed systems proposed in this thesis, we have integrated the *Expand* parallel file system into the BOINC software, so that the use of this file system is completely transparent to applications running in BOINC.

The transparent integration is made by a wrapper to the *Expand* parallel file system, so that when a job starts, the wrapper performs the following actions:

1.- The wrapper transfers the input data needed by the application from a pool of data servers in parallel.

2.- The data is never written to disk, instead the wrapper executes the application, passing the input data to the application through its standard input. The application processes the input data and write the results to its standard output. The output data produced by the application is captured by the wrapper through the standard output. As happens with the input data, the output data is never written to disk.

3.- The wrapper transfers the output data produced by the application in parallel directly to the distributed partition.

The main benefits of this new model are:

- The data are accessed in parallel from the data servers. This means a higher potential throughput between a client and the BOINC data server.
- The data is never written to disk, instead the data is passed to the application through its standard input/output. This is specially useful for clients with poor storage capacity. Because of data will not be in clients, sensible data does not need encryption for being stored on client disk.
- An application which is able to operate by reading the input data from its standard input and by writing the results to its standard output can work in this model without any change.

4.4.3 Results, analysis, and discussion of evaluation in real environments

The main motivation for our performance test is to study whether the new proposed model of transferring data from the data servers in parallel is feasible, and if it achieves better performance [Bergua et al. (2009a, 2010)]. With this aim, we have counted the number of work units successfully computed in one hour using the classical model of one BOINC server and the *Expand* solution using a parallel partition of four nodes. In each case we have chosen two kind of applications for the work units:

- One that does some processing with the input data ("Processing" in the figures). Input files are JPG images, and the processing consists of applying a smoothing filter to the image using the `convert` tool.
- And another that is data intensive and does no computing at all, just transfers data ("Data intensive" in the figures).

Both applications have been tested with two different kind of input/output files: a big file of 100 MB, and another smaller file of 4 MB.

The way the benchmark works is as follows: it creates a project of 1,000 work units with a 100 MB input file each, and another project of 20,000 work units with a 4 MB input file each. Once the projects has been created a shell script starts the BOINC client on the worker nodes in parallel and stops them one hour later. Finally, we count the number of work units successfully finished.

To perform this test we have used the following platform:

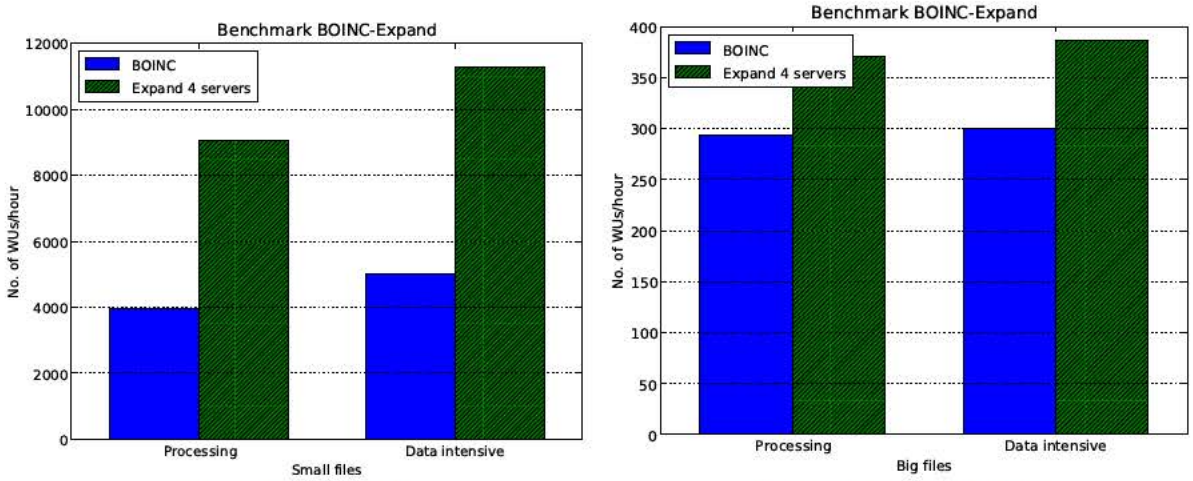
- A BOINC server and data repository: Intel® Xeon® CPU Quad Core, 4 GB RAM
- 4 workstations as standard servers for *Expand*: Intel® Core® 2 Quad CPU Q8200 @ 2.33GHz, 4 GB RAM.
- 68 workstations as clients taken from the Desktop Grid platform (AMD Athlon™ 64 X2 Dual Core Processor 4200+, 2 GB RAM).

All of them located in the campus of Leganés of Universidad Carlos III de Madrid.

Figures 4.28(a) and 4.28(b) show the number of work units successfully computed, obtained after running the benchmark for one hour. The former for small files and the latter for large

files. The benchmark proves that the proposed model (dark bar in the figures) achieves better performance than the traditional one (gray bar in the figures) with a higher number of work units processed by client per hour [Bergua et al. (2010)].

For large files, the work units throughput is more than 15% with our proposal. But for small files, the results are more than 200% better.



(a) Results for 4 MB files

(b) Results for 100 MB files

Figure 4.28: *Results of Expand in volunteer computing (4 MB and 100 MB files, Desktop Grid and Cluster platforms)*

As shown, there is a clear effect with the file size. To study deeper the effect of file size, and the number of data servers, a different benchmark has been developed. To better identify the effect of file size and number of servers, this new benchmark will not do any processing, just transfer data.

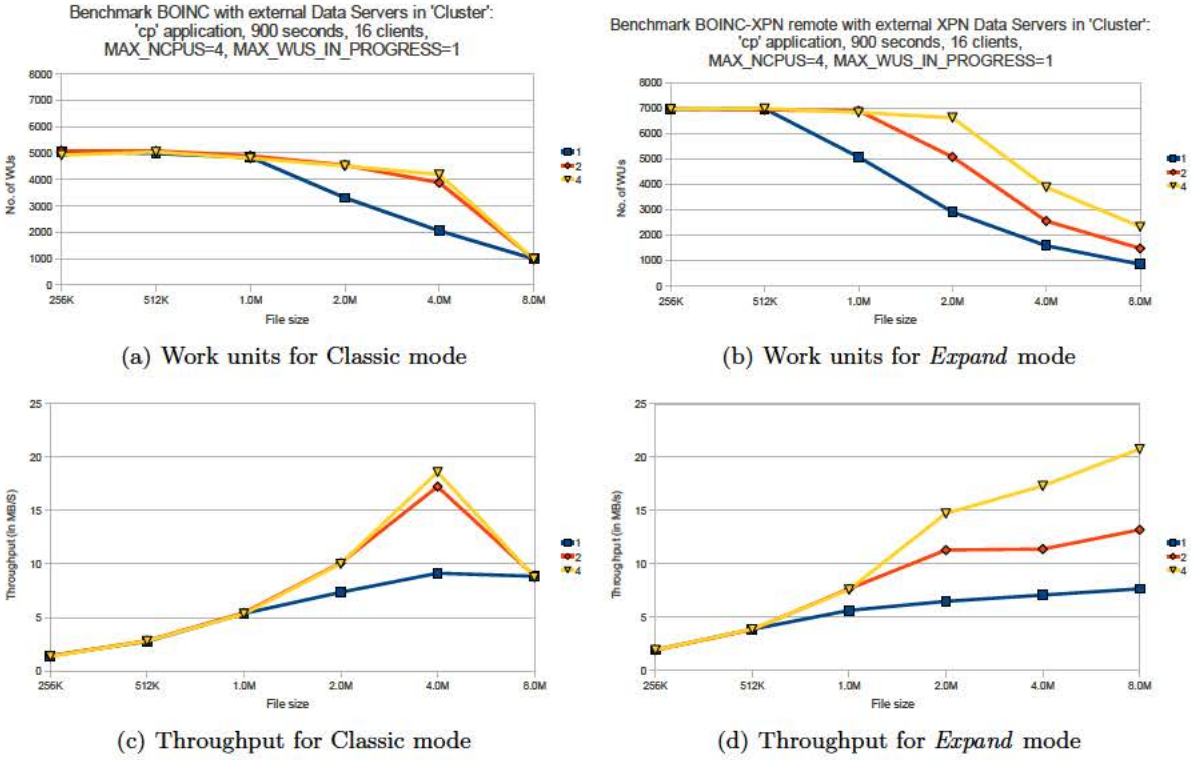
The benchmark creates a project of 10,000 work units with size of input files ranging from 256 KB to 8 MB. Once the project has been created a shell script starts 16 BOINC clients on the worker nodes in parallel and stops them fifteen minutes later. Finally, we count the number of work units successfully finished. The above process is repeated for 1, 2, and 4 data servers.

To perform this test we have used the following platform:

- 1 central BOINC server: 4xIntel® Xeon® CPU Quad Core, 16 GB RAM.
- 4 HTTP data servers: Intel® Pentium® 4 CPU Hyper-Threading, 2 GB RAM.
- 16 BOINC clients: Intel® Xeon® CPU Quad Core, 4 GB RAM.
- Clients are connected to central and data servers through two Gigabit Ethernet networks:
  - Clients $\rightleftarrows$ central BOINC server.
  - Clients $\rightleftarrows$ HTTP data servers.

Figures 4.29(a) and 4.29(c) show the number of work units successfully computed, and throughput respectively, in classic mode, used by traditional volunteer computing applications, obtained after running the benchmark for fifteen minutes.

Figures 4.29(b) and 4.29(d) show the number of work units successfully computed, and throughput respectively, with *Expand*, accessing the data in parallel and remotely, without writing data to disk, but processing them on-the-fly instead.



(a) Work units for Classic mode

(b) Work units for *Expand* mode

(c) Throughput for Classic mode

(d) Throughput for *Expand* mode

Figure 4.29: *Results of Classic and Expand modes in volunteer computing by file size (Desktop Grid and Cluster platforms)*

These results show that there is a significant benefit when using *Expand* in volunteer computing environments for file sizes greater or equal than 1 MB. In the worst case *Expand* performs as good as the traditional approach, but the greater the file size, and the more the number of servers, the bigger the benefit. Results show that the performance improvement can surpass 200% [Bergua et al. (2009a, 2010)].

4.5 Summary

This chapter has presented the main results of the evaluation of the generic I/O middleware proposed in this thesis: a parallel file system for data access in large-scale distributed systems, and its application in Grid and volunteer computing environments. Results show that there is a significant improvement when using the generic I/O middleware architecture proposed compared to the traditional approach.

However, the proposed architecture does not take into consideration small transfers. Therefore, an algorithm that takes care of small transfers, as well as big transfers, is needed to really be a generic architecture.

Chapter 5

Optimizations for replica selection in large-scale distributed systems

This chapter presents the optimizations for replica selection designed for large-scale distributed systems. The rest of this chapter is organized as follows. First, the Motivation and objectives of these optimizations will be explained, later the Problem definition is stated, then the Selection of virtual parallel partitions in large-scale replicated environments is explained, and finally, the Evaluation of the replica selection algorithm in Grid and volunteer computing environments is presented.

5.1 Motivation and objectives

This chapter aims to fulfill the second primary objective indicated in Section 1.2: to design a **replica selection algorithm** for configuring access virtual parallel partitions for large-scale distributed systems.

There are many works related to the topic of replica selection in replicated environments. Basically, all works that focus on replica selection pursue two objectives:

1.- Optimize the download of whole files

2.- Optimize the download of big files

However, besides the previous problems, in large-scale distributed systems there are, also, additional problems or needs:

- Need to access to data, but not necessarily write them to disk
- Need to access to just some (possibly small) part of a file, but not necessarily the whole file

- Need to access to or download small files

The objective of this chapter is, thus, to propose a new approach for accessing to file replicas to optimize the access to virtual parallel partitions configured basing on the available set of replicas for a given file.

Now, let us suppose that among a hypothetical list of data servers, some of them are clearly better than the rest in terms of latency and throughput, i.e., they offer low latency and high throughput. An obvious strategy for replica selection would suggest to select these subset of data servers. However, if we always select the same subset of data servers because they have reported to be good, they will quickly become a bottleneck. A replica selection algorithm should assure a good balance among server's workloads, while providing high overall performance in the system.

Many approaches to this problem suggest to constantly monitor the status of the network to adjust the configuration dynamically. However, this solution introduces extra complexity and overhead in the network.

Ideally, the objectives of a replica selection algorithm for large-scale distributed systems are:

- Optimize the transfer of chunks of data ranging from tiny chunks to huge files
- Take advantage of all kinds of data servers, ranging from bad servers (high latency, low throughput) to good servers (low latency, high throughput)
- Balance workload on servers proportionally to their characteristics
- Select the best level of parallelism for any given request
- Be fault tolerant, so that if all nodes but one die, the transfer still can take place
- Be as autonomous as possible, getting rid of agents or other external systems, that introduce complexity and overhead

The objective of this chapter is to design, and evaluate, an algorithm for configuring virtual parallel partitions, given a list of replicas for a given file, and based on a probability function that chooses the servers or set of servers randomly among all the servers that hold a replica of a given file, and creating a virtual parallel partition to transfer the file in parallel from this random set of servers. The set of servers used by the algorithm for creating a virtual parallel partition is chosen randomly among all the servers that hold a replica of the file that needs to be transferred. The way the algorithm chooses the set of servers randomly is by using a probability function that gives to each server a probability of being chosen, based on a weight function that gives to every server a weight depending on how well suited is each server for a given transfer. The weight function gives a weight to every server for a given transfer, prioritizing low latency servers for small transfers, and high throughput servers for big transfers.

5.2 Problem definition

Given a list of servers S, defined as:

$$S = \bigcup_{i=1}^{k} s_i$$

And given a set of files F, defined as:

$$F = \bigcup_{i=1}^{n} f_i$$

where each file f_k can be replicated in a set S_{f_k} of 1 or more servers, being S_{f_k} defined as:

$$S_{f_k} \subseteq S$$

The objective is to define an access virtual parallel partition $AVPP$, defined as:

$$AVPP \subseteq S_{f_k}$$

that allows to access to this file using the generic I/O middleware proposed in this thesis.

Classic approaches to access to replicated data use replica selection algorithms to access to the best set of replicas to download a file, or set of files, usually big files, as fast as possible.

However, the approach proposed in this thesis pursues to create a virtual parallel partition ($AVPP$), chosen among the servers (S_{f_k}) that hold a replica of a given file (f_k), which is optimized for a given size access. This means, that an $AVPP$ for a small transfer of a given file f_k will be possibly different than an $AVPP$ for a bigger transfer of the same file f_k. Each time, the $AVPP$ is chosen to optimize the specific transfer that is going to take place.

Traditionally, the basic equation for predicting how long a transfer will last is shown in Equation 5.1. Many approaches use this equation as the reference equation for selecting the best server or set of servers for doing a transfer, since it does a very good job for predicting the time of big transfers.

$$\text{time} = \text{latency} + \frac{\text{file size}}{\text{throughput}} \tag{5.1}$$

However, if we use Equation 5.1 as the reference equation for a generic I/O middleware architecture for large-scale distributed systems, the best servers (those with low latency, and high throughput) will always be the best candidates for every transfer. In big transfers, the throughput component hides the latency. In a general purpose large-scale distributed environment, in which transfer sizes range from tiny transfers to huge transfers, the best servers will quickly become overloaded by big transfers. And this would cause a terrible penalty to small transfers.

Thus, we need to separate both latency and throughput components, and give them different treatments when selecting the server or set of servers for any given transfer, giving them different weights depending on the transfer size.

A good replica selection algorithm for a generic I/O middleware architecture for large-scale distributed systems would need to give special importance to low-latency servers for small transfers, and high-throughput servers for big transfers (see Table 5.1).

	Latency	Throughput
Small transfers	High	Low
Big transfers	Low	High

Table 5.1: *Summary of importance of latency and throughput*

5.3 Selection of virtual parallel partitions in large-scale replicated environments

We need to design a sort of mechanism, so that small transfers will have higher priority in the lowest latency servers, and big transfers will be have higher priority in the highest throughput servers. In order to properly model the requirements of table 5.1, we need:

- For weighting the latency: a function that must be increasing when file size tends to 0, and decreasing when file size gets arbitrary big.
- For weighting the throughput: a function that must be decreasing when file size tends to 0, and increasing when file size gets arbitrary big.

With the above needs in mind we propose:

- $y = \frac{1}{x}$ as the base function for weighting the latency (being x the file size)
- $y = e^x$ as the base function for weighting the throughput (being x the file size)

Figure 5.1 shows the shape of the two proposed functions for the algorithm.

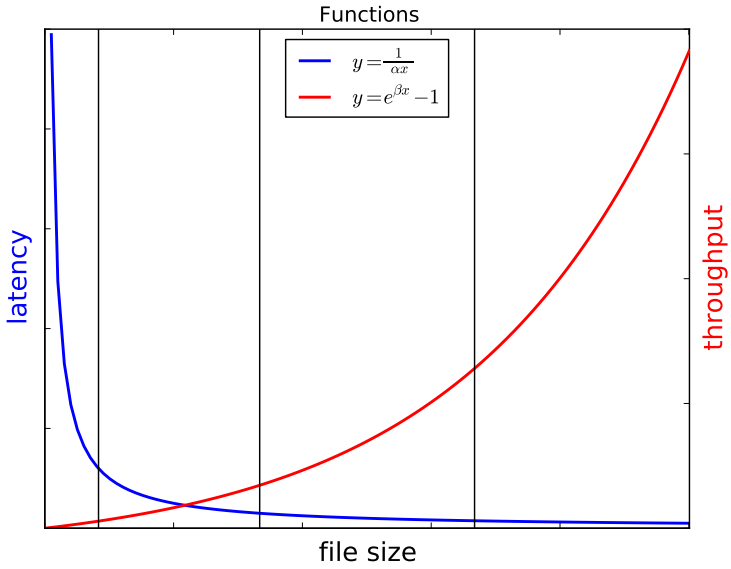


Figure 5.1: *Functions*

The black vertical lines represent some examples of different file sizes, and the cut points between the black vertical lines and the weighting functions (blue and red lines) are the specific weights that the weighting functions will have for those file sizes. Final weights will be a combination of those two cut points for every black line.

As can be seen, the two proposed functions for weighting latency and throughput hold that, if used in combination, they would give higher importance to low-latency servers in small transfers, and to high-throughput servers in big transfers, i.e., they fulfill the needs of Table 5.1. Figure 5.2 shows some specific examples of how these functions would behave for different file sizes.

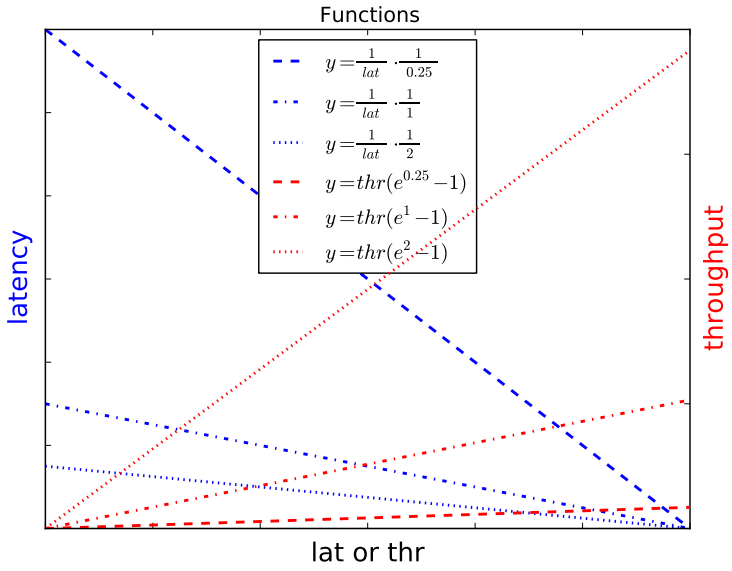


Figure 5.2: *Weighting functions: some examples*

Equations 5.2 and 5.3 show the final expressions of the weighting functions. Some terms have been added to the original expressions of Figure 5.1 to make them work in corner cases.

$$w_{lat}(fs) = \frac{1}{lat+1} \cdot \frac{1}{\alpha fs} \tag{5.2}$$

$$w_{thr}(fs) = thr(e^{\beta(fs-1)} - 1) \tag{5.3}$$

Being α and β the slope parameters that can be tuned for optimal performance, for example, using some optimization method to find the pair of values that gives the best performance.

Though intuitively these functions fulfill the needs, let us do a theoretical study of these functions in corner cases. Table 5.2 shows $\lim_{fs\to 1} w_{lat}$ and $\lim_{fs\to 1} w_{thr}$:

$\lim_{lat \to 0} w_{lat} = \frac{1}{\alpha}$	$\lim_{thr \to 1} w_{thr} = 0$
$\lim_{lat \to lat_{max}} w_{lat} = \frac{1}{lat_{max}+1} \cdot \frac{1}{\alpha}$	$\lim_{thr \to thr_{max}} w_{thr} = 0$

Table 5.2: $\lim_{fs \to 1} w_{lat}$ *and* $\lim_{fs \to 1} w_{thr}$

Which holds that $w_{lat=0} > w_{lat} > w_{lat_{max}} \quad \forall \quad 0 < lat < lat_{max}$. In particular, if $fs = 1$:

If $lat = 0$, $w_{lat} = \frac{1}{\alpha}$
If $lat = lat_{max}$, $w_{lat} = \frac{1}{lat_{max}+1} \cdot \frac{1}{\alpha}$
$w_{thr} = 0 \quad \forall \quad thr$

Table 5.3: w_{lat} *and* w_{thr} *for* $fs = 1$

Table 5.4 shows $\lim_{fs \to \infty} w_{lat}$ and $\lim_{fs \to \infty} w_{thr}$:

$\lim_{lat \to 0} w_{lat} = 0$	$\lim_{thr \to 0} w_{thr} = 0$
$\lim_{lat \to lat_{max}} w_{lat} = 0$	$\lim_{thr \to thr_{max}} w_{thr} = \infty$

Table 5.4: $\lim_{fs \to \infty} w_{lat}$ *and* $\lim_{fs \to \infty} w_{thr}$

Which holds that $w_{thr=0} < w_{thr} < w_{thr_{max}} \quad \forall \quad 0 < thr < thr_{max}, \quad 1 < fs < \infty$. In particular, if $1 < fs < \infty$:

If $lat = 0$, $w_{lat} = \frac{1}{\alpha fs}$	If $thr = 0$, $w_{thr} = 0$
If $lat = lat_{max}$, $w_{lat} = \frac{1}{lat_{max}+1} \cdot \frac{1}{\alpha fs}$	If $thr = thr_{max}$, $w_{thr} = thr_{max}(e^{\beta(fs-1)} - 1)$

Table 5.5: w_{lat} *and* w_{thr} *for* $1 < fs < \infty$

Equation 5.4 shows the final expression of the combination of the two weighting functions as a function of file size.

$$w_i(fs) = \theta w_{lat}(fs) + (1 - \theta) w_{thr}(fs), \quad 0 \leqslant \theta \leqslant 1 \tag{5.4}$$

Being θ the balance parameter between latency and throughput. Finally, Equation 5.5 represents Equation 5.4 turned into a probability function. It gives the probability of server i of being chosen for a given file size.

$$p_i = \frac{w_i}{\sum_{j=1}^{n} w_j} \tag{5.5}$$

The final algorithm must be divided in two steps. The first step consists of locating the list of servers that hold a replica of the given file. This step must be done when opening the file (see Algorithm 5.1).

The second step consists of calculating weights for all servers that hold a replica based on the transfer size requested, and then converting the weights into probabilities. Finally, a set of

```
1  open(filename, flags [, mode]) {
2          for each S_i ∈ S
3                  for each SU_ij ∈ S_i
4                          if filename ∈ SU_ij
5                                  S_fd ← SU_ij
6          // Continue with the rest of operations for opening a file
7          f_M(filename) → SU_M ∈ S_fd
8          for each SU_i ∈ S_fd
9                  {obtain fh_i ∈ S_i}
10         {obtain S_M ∈ SU_M}
11         fd ← FH
12 }
```

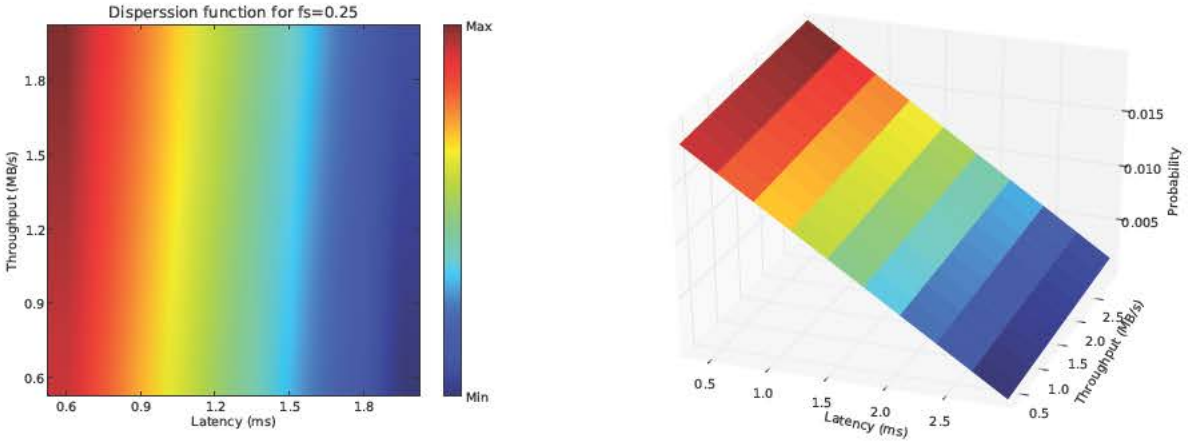
Algorithm 5.1: *File opening operation in the replica selection algorithm*

servers is chosen randomly. This step must be carried out every time a transfer on the given file is requested, i.e., in every read request (see Algorithm 5.2).

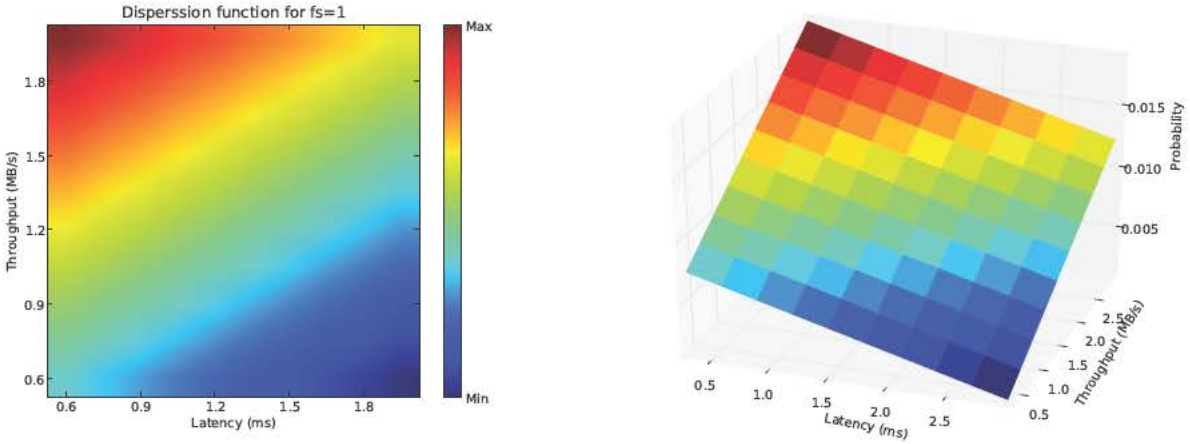
```
1  read(fd, buffer, size) {
2          {obtain S_fd of fd}
3          for each S_i ∈ S_fd
4                  compute w_lat_i(size) (Equation 5.2)
5                  compute w_thr_i(size) (Equation 5.3)
6                  compute w_i (Equation 5.4)
7          for each w_i
8                  compute p_i (Equation 5.5)
9          AVPP ← random({p_i, SU_i}∀SU_i ∈ S_fd)
10         // Continue with the rest of operations for reading a file
11         for each SU ∈ AVPP
12                 if fh_i ∩ SU_i = ∅
13                         {obtain fh_i}
14         {divide buffer in {d, v} whose size ≤ ||B_ij|| ∈ AVPP }
15         ∀{d, v} ∈ buffer
16                 f_d(d_k, v_k) ⇐ B_ij ∈ S_i
17 }
```

Algorithm 5.2: *Parallel read operation in the replica selection algorithm*

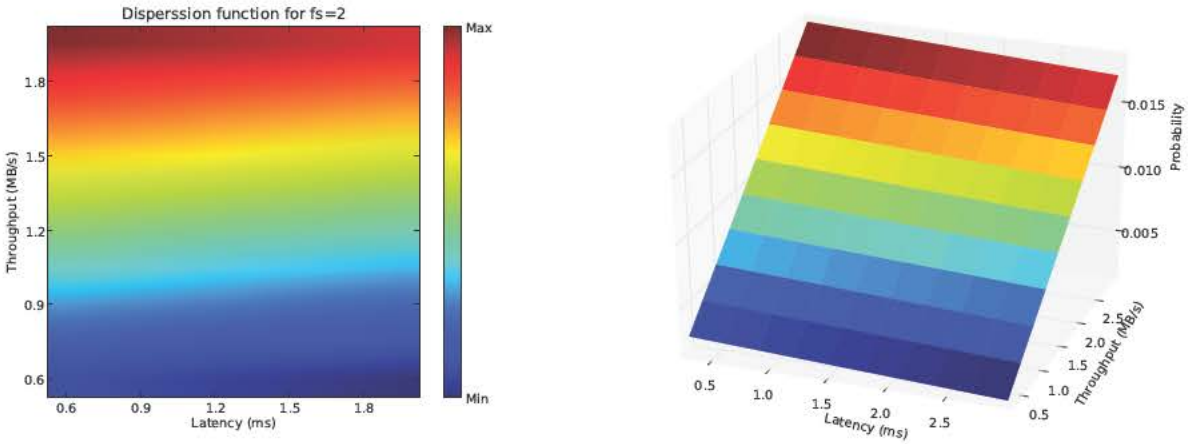
Figure 5.3 shows 2D and 3D colored representations of the behavior of the final Equation 5.5 for different fake examples of file sizes. It shows how, when the file size is small, the lowest latency servers have the highest weights; and as file size increases, the weights turn to the highest throughput servers.



(a) fs small



(b) fs medium



(c) fs large

Figure 5.3: *Disperssion functions*

5.4 Evaluation of the replica selection algorithm in Grid and volunteer computing environments

This section presents the main results of the evaluation of the optimizations proposed in this thesis: a replica selection algorithm for configuring virtual parallel partitions in large scale distributed systems, and its application for Grid and volunteer computing environments.

5.4.1 Simulation environment

Performing tests for large-scale scenarios requires access to big platforms, and in case access to those platforms is granted experiments might take long time. To ease the task of evaluating the proposed solutions a simulator of distributed I/O has been developed. Next sections explain the components of the simulator, and its validation.

5.4.1.1 Components of a distributed I/O simulator

We have chosen **SimGrid** as the simulation framework because it is extremely scalable (simulations of more than 2 million nodes have been reported to work [Quinson et al. (2012)]) while accurate, and provides abstractions for building distributed system simulators: network (send/recv, wait/waitall/test/listen/etc.), and tasks (exec/pause/resume/kill). The simulator developed for this thesis is built on top of **SimGrid v3.10**. SimGrid has been extensively validated by its authors (see Section 2.7.3 SimGrid for details about SimGrid).

Data server The data server is responsible of accepting download/upload requests from clients. When a download request is accepted, it reads the file from disk, and sends it to the client. The data server can serve whole files, or part of them. When an upload request is accepted, it receives the file from the client, and once this transfer has finished the server writes the file to disk.

Disk A disk is simulated using the same simulation model as that used for simulating how tasks execute on CPUs in SimGrid. Though intuitive, we do not use network links to simulate access to disks because links have some side effects to accurately simulate TCP under congestion. On the contrary, the simulation model of the execution of tasks in CPUs in SimGrid is a fair sharing model, what means that if two tasks request access to disk, each task will obtain half of the disk throughput, without network-related side effects.

Grid client Grid clients request files to data servers. They can request whole files or part of them to a single data server, or to several data servers in parallel. Once the file is received, the client can write it to disk for later processing, or process it on-the-fly without making use of the disk.

VC client A volunteer computing client is very similar to a grid client, but a VC client have a predefined script of functioning that repeats over and over again:

1.- Request a job to a VC server.

2.- Download input files from one data server, or from several in parallel.

3.- Write input files to disk for later processing, process the files and write the results to disk. Or process them on-the-fly.

4.- Upload results to one data server, or to several in parallel.

5.- Inform the server that the job has been completed.

VC server A volunteer computing server is responsible of accepting requests from VC clients, and assign jobs to clients. Whenever a VC server receives a job request from a client, it chooses the next unfinished job, and a set of one or several data servers to download the input files from, and sends the chosen job along with the set of servers to the client.

5.4.2 Workload modeling

In order to evaluate the proposed algorithm we need a realistic workload of files with different sizes. A good and realistic workload is important to draw conclusions about the benefits of the proposed algorithm. This section will explore the complexity of the Characterization of file size distributions, and later the Benchmark workloads will be explained.

5.4.2.1 Characterization of file size distributions

[Feitelson (2015)] points out that the distribution of file sizes in a file system or retrieved from a web server follows a heavy-tailed distribution [Irlam (1993), Crovella and Bestavros (1997), Barford and Crovella (1998), Downey (2001)]. So does the distribution of popularity of items on a web server, and the distribution of popularity of web sites [Barford and Crovella (1998), Breslau et al. (1999), Roadknight et al. (2000), Adamic and Huberman (2001), Oke and Bunt (2002)].

Heavy tails are most commonly defined as those governed by power laws. What follows is some basic definitions of power law, Pareto, and lognormal distributions taken from [Mitzenmacher (2003a,b)].

A non-negative random variable X is said to have a *power law distribution* if the *complementary cumulative distribution function* (ccdf), or $\Pr[X > x]$, satisfies

$$\Pr[X > x] \sim cx^{-\alpha}$$

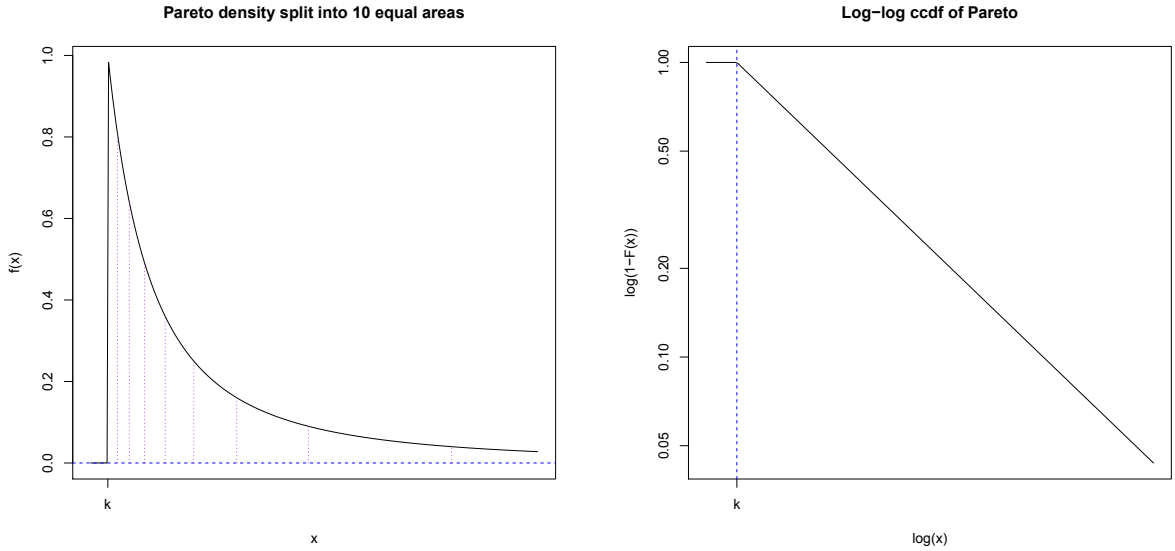
for constants $c > 0$ and $\alpha > 0$. Here, $f(x) \sim g(x)$ denotes that the limit of the ratios goes to 1 as x grows large. One specific commonly used power law distribution is the *Pareto distribution*, which satisfies

$$\Pr[X > x] \sim \left(\frac{x}{k}\right)^{-\alpha}$$

for some $\alpha > 0$ and $k > 0$. The *probability density function* (pdf) for the Pareto distribution is

$$f(x) = \alpha k^{\alpha} x^{-\alpha-1}$$

If X has a power law distribution, then in a log-log plot of the ccdf, asymptotically the behavior is a straight line. See Figures 5.4(a) and 5.4(b) for a graphical representation of the pdf and ccdf of a Pareto distribution.



(a) Pareto pdf

(b) Pareto ccdf (log-log)

Figure 5.4: *Pareto distribution*

A random variable X has a *lognormal distribution* if the random variable $Y = \ln X$ has a normal (i.e. Gaussian) distribution. The density function for a lognormal distribution satisfies

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} e^{-(\ln x - \mu)^2/2\sigma^2}$$

where μ is the mean and σ is the standard deviation of the associated normal distribution. See Figure 5.5 for a graphical representation of the pdf of a lognormal distribution.

The lognormal distribution is extremely similar in shape to power law distributions, in the following sense: if X has a lognormal distribution, then in a log-log plot of the ccdf or the density function, the behavior will be a straight line except for a large portion of the body of the distribution. See Figures 5.6(a) and 5.6(b) for a graphical representation of the pdf and ccdf of a lognormal distribution.

[Barford and Crovella (1998)] proposed a model for file sizes based on a lognormal distribution for the body and a Pareto for the tail. This model for file sizes and other aspects of workload characterization are included in SURGE, a web workload generation tool. A summary of distributions and their parameters regarding file and request sizes in SURGE are shown in Table 5.6:

These results are based on previous work by [Arlitt and Williamson (1996), Crovella and Bestavros (1997)], whose basic assumptions were later confirmed by [Williams et al. (2005)].

Up to 2001 there was broad consensus on that the distribution of file sizes follows a lognormal distribution for the body and a Pareto for the tail. However, [Downey (2001)] found no evidence for long tails (the Pareto distribution is considered long-tailed). Instead, he proposed a model for the evolution of a file system over time, based on a multiplicative process. The idea is that the size of a new file can be modeled by taking the size of an old file and multiplying it by a

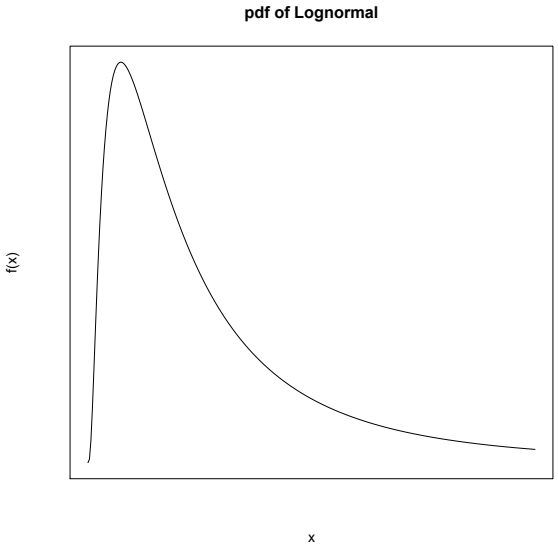
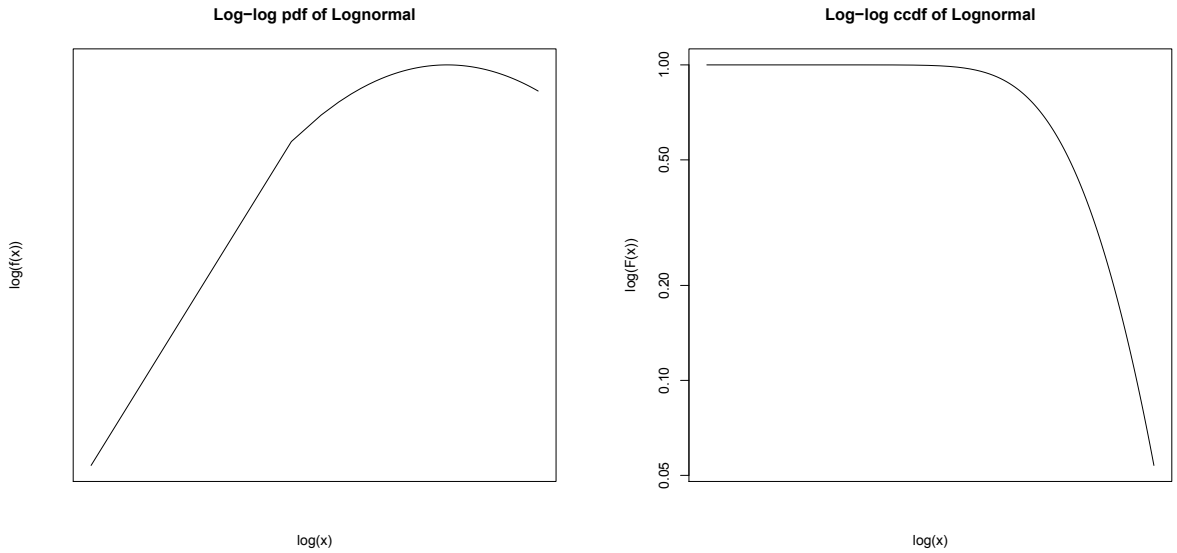


Figure 5.5: *Lognormal distribution*



(a) Lognormal pdf (log-log)

(b) Lognormal ccdf (log-log)

Figure 5.6: *Lognormal distribution (log-log)*

random variable. Downey argued that this model yields a lognormal distribution for file sizes. If a model with more parameters is needed, it can be extended to include more than one mode with a two-mode lognormal. He concluded that long-tailed models are not necessary to describe the observed distributions, and that his user model is a better fit for the data than the hybrid lognormal-Pareto model [Downey (2001)].

Lognormal distributions can be naturally generated by multiplicative processes. Lognormal

Component	Model	Probability Density Function	Parameters
File sizes - Body	Lognormal	$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-(\ln x - \mu)^2/2\sigma^2}$	$\mu = 9.357$, $\sigma = 1.318$
File sizes - Tail	Pareto	$p(x) = \alpha k^{\alpha} x^{-(\alpha+1)}$	$k = 133\text{K}$, $\alpha = 1.1$
Popularity	Zipf	$p(x) = kx^{-s}$	
Request sizes	Pareto	$p(x) = \alpha k^{\alpha} x^{-(\alpha+1)}$	$k = 1000$, $\alpha = 1.0$

Table 5.6: *Summary statistics for models used in* SURGE *(1998)*

distributions are natural for describing growth of organisms, and any process where over a time-step the underlying growth is a random factor independent of the current size [Mitzenmacher (2003a,b)].

Then, [Mitzenmacher (2003b)] expanded Downey's work to a dynamic model that allows additions (not necessarily derived from existing files, as opposed to Downey's model) and deletions. As a result he obtained a family of models referred to as Recursive Forest File model. The resulting distribution of file sizes is a double Pareto distribution. Double Pareto distributions have been suggested to describe several power law phenomena [Reed (2003), Reed and Jorgensen (2004)]. These distributions have a lognormal body and a Pareto tail, which matches some previous studies of empirical data for file sizes [Mitzenmacher (2003b)].

To understand how a power law can be obtained from a lognormal distribution, I present a summarized explanation taken from [Mitzenmacher (2003b)]. Suppose we have a system $X_t = F_t X_{t-1}$, where $X_0 = 1$ and F_t is a lognormal distribution with parameters (μ, σ^2). Think of index t as referring to time. If we let the system run and stop it at some fixed time k, we obtain a random variable from the lognormal distribution with parameters $(k\mu, k\sigma^2)$. Suppose instead we run the system until some random time k. Then we obtain a random variable that comes from a mixture of lognormal distributions. Specifically consider the case where we have a geometric mixture of lognormal distributions. Then, the resulting distribution from this mixture will have a power law. Exponential mixtures can be used instead of geometric mixtures for convenience. I refer the interested reader to [Mitzenmacher (2003a,b), Mitzenmacher and Tworetzky (2003)] for further details on the above explanation, and empirical studies.

A *double Pareto distribution* defined over $x > 0$ with parameters $\alpha, \beta > 0$ has pdf

$$f(x) = \begin{cases} \frac{\alpha\beta}{\alpha+\beta} x^{\beta-1}, & \text{for } 0 < x \leq 1 \\ \frac{\alpha\beta}{\alpha+\beta} x^{-\alpha-1}, & \text{for } x > 1 \end{cases}$$

See Figure 5.7 for a graphical representation of the pdf of a double Pareto distribution.

A key characteristic of the double Pareto distribution is that it has a power law at both tails. That is, if we look at the *cumulative distribution function* (cdf) on a log-log plot, it will also have a linear tail (for the small files). This provides a test for seeing whether a distribution has a double Pareto distribution; look at both the ccdf and the cdf on log-log plots for linear tails [Mitzenmacher (2003b)]. See Figures 5.8(a) and 5.8(b) for a graphical representation of the cdf and ccdf of a double Pareto distribution.

The double Pareto distribution falls nicely between the lognormal distribution and the Pareto distribution. Like the Pareto distribution, it is a power law distribution. But while the log-log plot of the density of the Pareto distribution is a single straight line (see Figure 5.9(a)), for the double

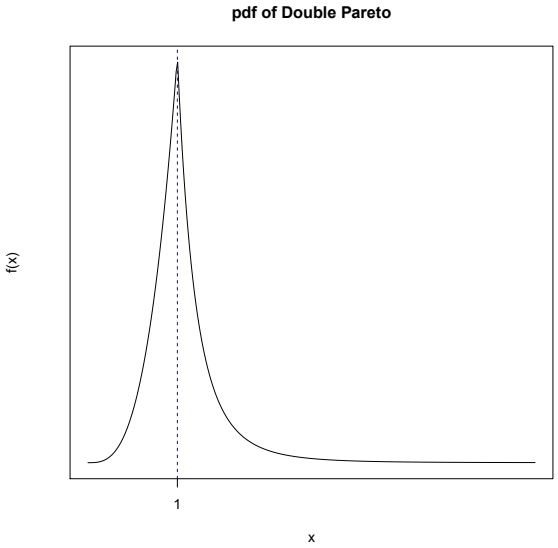
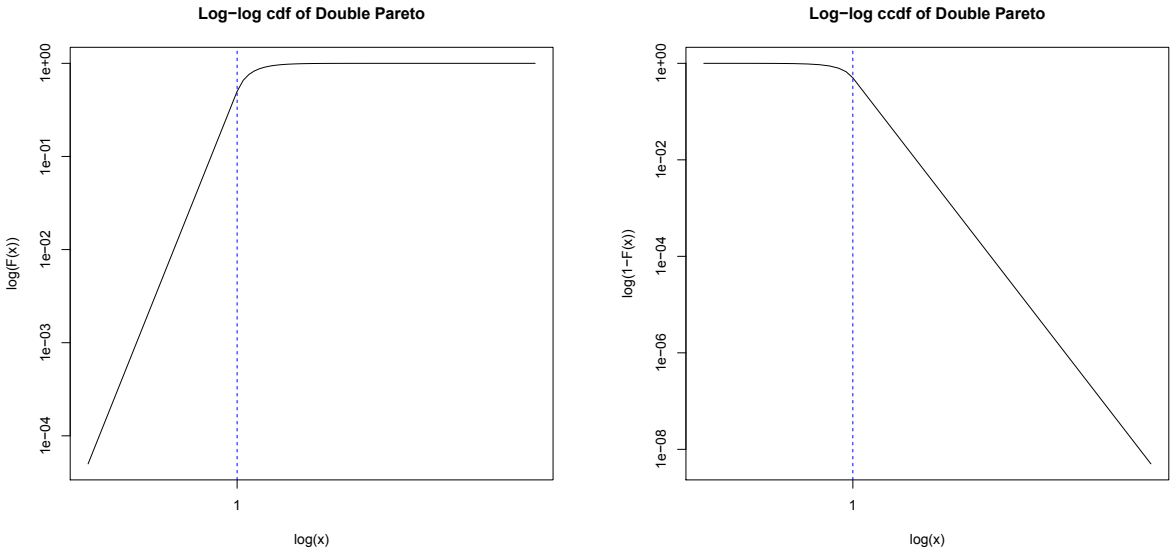


Figure 5.7: *Double Pareto distribution*



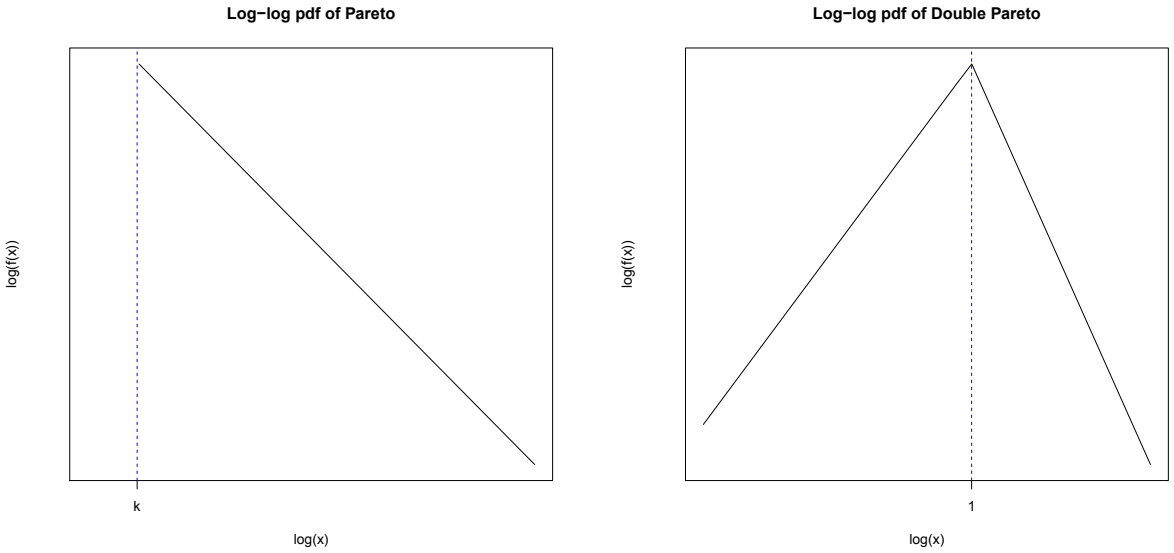
(a) Double Pareto cdf (log-log)

(b) Double Pareto ccdf (log-log)

Figure 5.8: *Double Pareto distribution (log-log)*

Pareto distribution the log-log plot of the density consists of two straight line segments that meet at a transition point (see Figure 5.9(b)). This is similar to the lognormal distribution, which has a transition point around its median e^{μ}. Hence, an appropriate double Pareto distribution can closely match the body of a lognormal distribution and the tail of a Pareto distribution. The ccdf for a lognormal and a double Pareto distributions match quite well with a standard scale for probabilities. On the log-log scale, however, the double Pareto distribution follows a power

law (see Figure 5.8(b)), while the lognormal distribution has a clear curvature (see Figure 5.6(b)) [Mitzenmacher (2003a,b)].



(a) Pareto pdf (log-log)

(b) Double Pareto pdf (log-log)

Figure 5.9: *Pareto-Double Pareto comparison (log-log)*

Also, Reed has suggested a generalization of the double Pareto distributions called double Pareto-lognormal distributions with similar properties [Reed and Jorgensen (2004)]. The double Pareto-lognormal distribution has more parameters, but might allow closer matches with empirical distributions [Mitzenmacher (2003b)].

The pdf of a *double Pareto-lognormal distribution* can be expressed in terms of the cdf Φ and ccdf Φ^c of $N(0,1)$ as

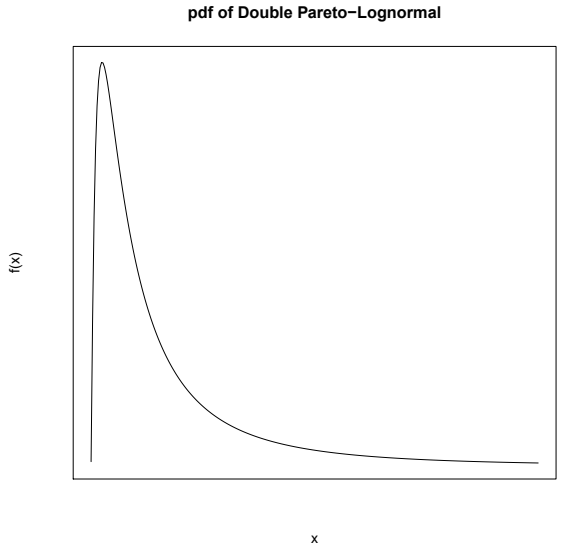
$$f(x) = \frac{\alpha\beta}{\alpha+\beta}\left[A(\alpha,\nu,\tau)x^{-\alpha-1}\Phi\left(\frac{\log x-\nu-\alpha\tau^2}{\tau}\right) + A(-\beta,\nu,\tau)x^{\beta-1}\Phi^c\left(\frac{\log x-\nu+\beta\tau^2}{\tau}\right)\right]$$

where

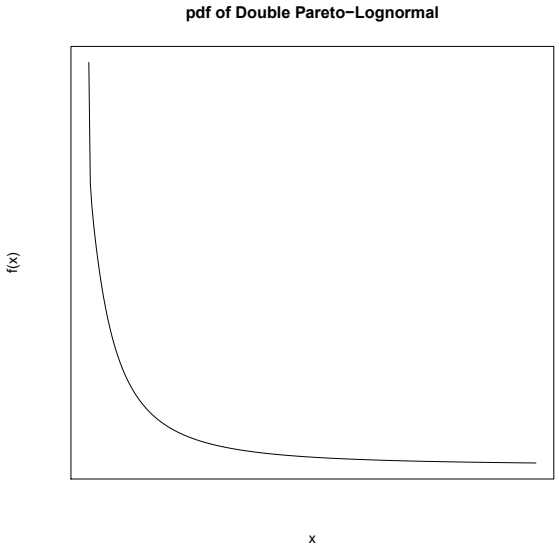
$$A(\theta,\nu,\tau) = \exp\left(\theta\nu + \frac{\theta^2\tau^2}{2}\right)$$

See Figure 5.10 for a graphical representation of the pdf of a double Pareto-lognormal distribution.

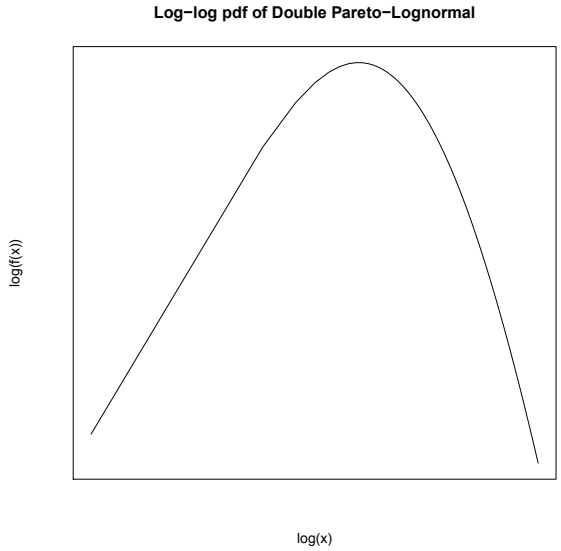
I refer the reader to [Reed (2001), Reed and Hughes (2002), Reed (2003), Reed and Jorgensen (2004)] for further details about power law, Pareto, double Pareto, and double Pareto-lognormal distributions, and their use to model size distributions.



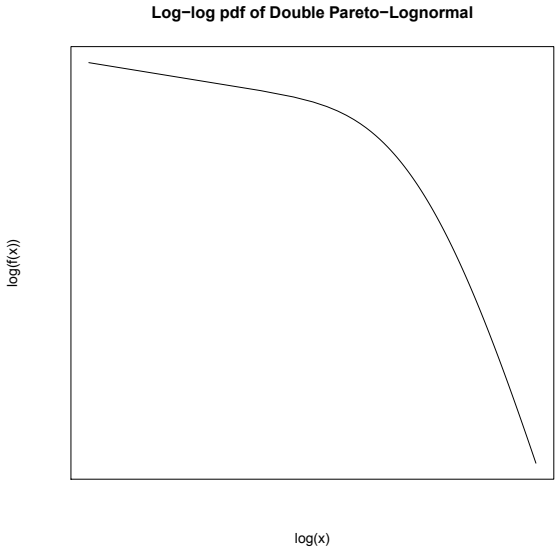
(a) Double Pareto-Lognormal pdf ($\beta > 1$)



(b) Double Pareto-Lognormal pdf ($\beta < 1$)



(c) Double Pareto-Lognormal pdf log-log ($\beta > 1$)

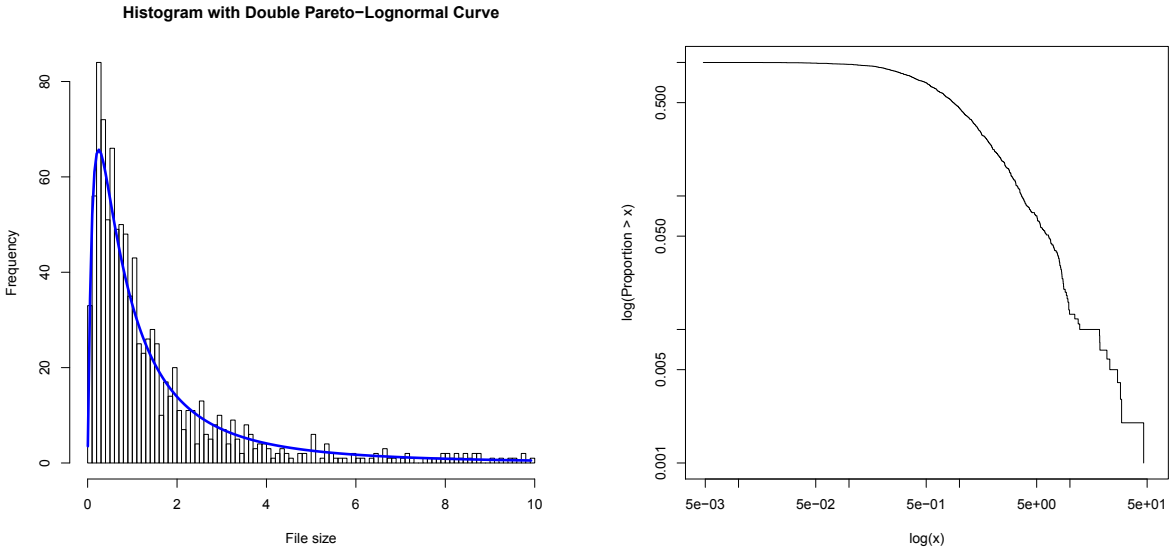


(d) Double Pareto-Lognormal pdf log-log ($\beta < 1$)

Figure 5.10: *Double Pareto-Lognormal distribution*

5.4.2.2 Benchmark workloads

According to the study of the characterization of file size distributions presented in Section 5.4.2.1, a double Pareto-Lognormal distribution, with parameters $\alpha = 1$, $\beta = 2$, $\nu = 0$, and $\tau = 1$, will be used. The number of files generated will be 1000. See Figure 5.11 for histogram, and *empirical complementary cumulative distribution function* (eccdf) of the generated sample.



(a) Double Pareto-Lognormal histogram

(b) Double Pareto-Lognormal eccdf

Figure 5.11: *Double Pareto-Lognormal sample*

Also, to study the impact of the solution, several units will be considered for the generated sample. In other words, different data sets will be considered using the same sample of 1000 files generated using the distribution specified above, the only difference among these data sets is the unit of the file sizes. Two different data sets will be considered for file sizes:

- FS-KB: 18 bytes → 5.05 MB
- FS-MB: 18.62 KB → 5.05 GB

This means that the first data set considered will be that in which each file size ranges from 18 bytes to 5.05 MB; and, the second data set is that in which each file size will ranges from 18.62 KB to 5.05 GB.

The above workload will be tested for a number of clients and servers:

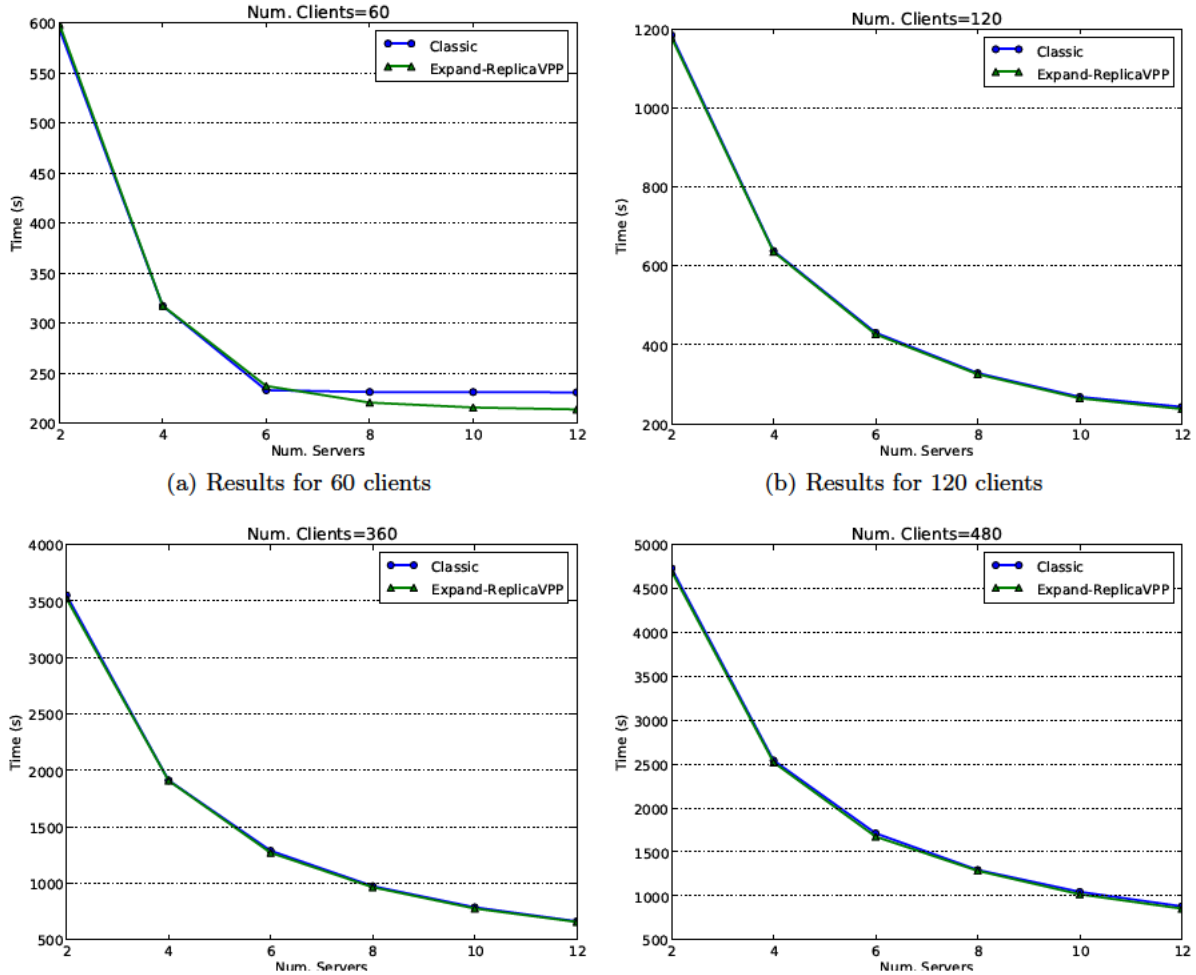
- Number of clients: 60, 120, 360, and 480
- Number of servers: 2, 4, 6, 8, 10, and 12

5.4.2.3 Platform definition

The experiments conducted in this chapter will use the simulator described in Section 5.4.1 Simulation environment. The platform simulated will be Grid5000, the Medium Grid platform.

5.4.3 Evaluation in Grid environments

Figure 5.12 shows results for 60, 120, 360, and 480 clients for the FS-KB dataset in Grid environment.



(a) Results for 60 clients

(b) Results for 120 clients

(c) Results for 360 clients

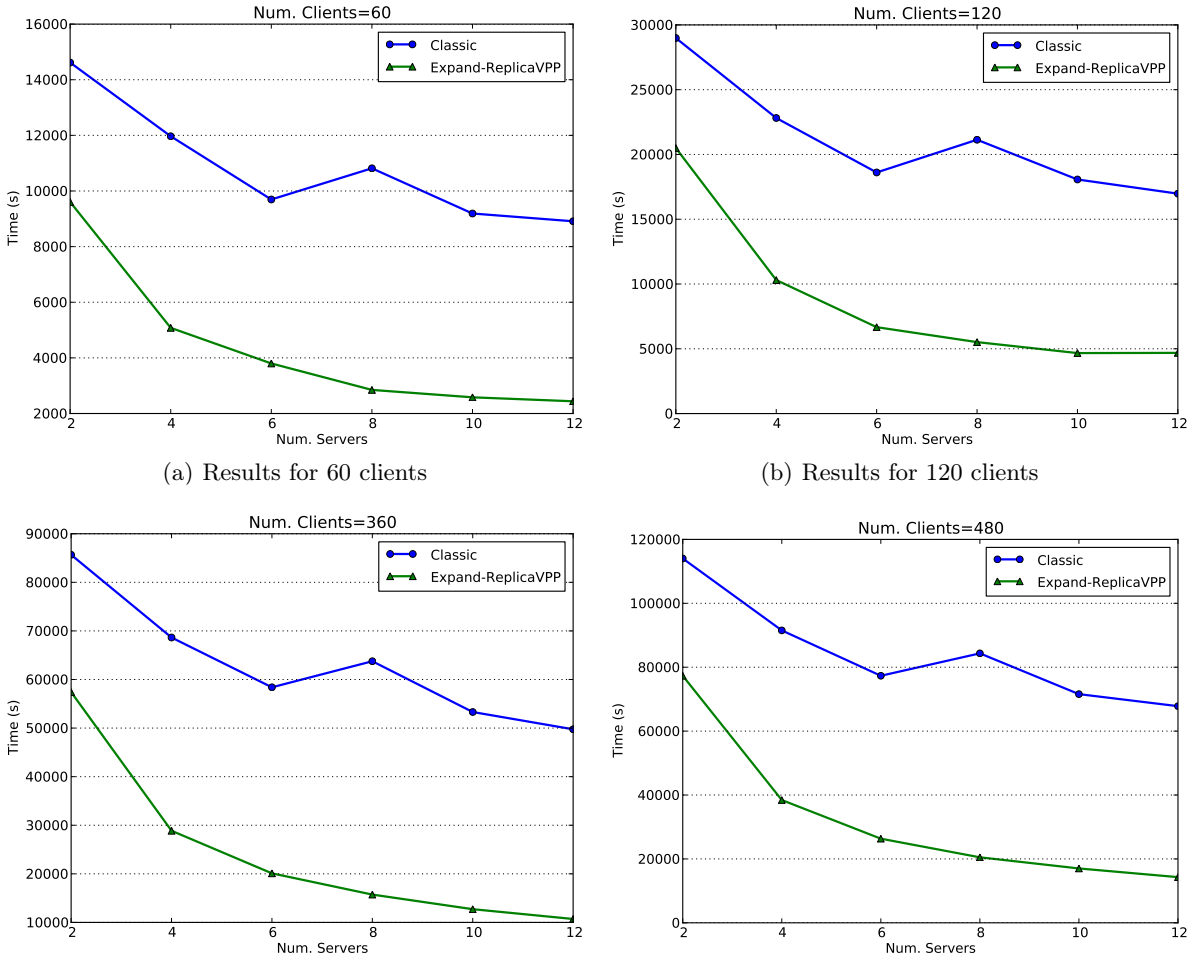
(d) Results for 480 clients

Figure 5.12: *Results of Classic vs Replica Selection Algorithm in Grid environment (FS-KB dataset, simulated Medium Grid platform)*

For a dataset like FS-KB with very small file sizes, the algorithm for replica selection shows marginal improvement over classical round-robin algorithm. The reason is that with such small files the underlying network and the servers are not saturated enough to appreciate the benefit of selecting the replicas with the algorithm proposed based on file size.

However, when the file sizes range from few kilobytes up to gigabytes, results show a significant improvement when using the replica selection algorithm proposed (see Figure 5.13).

In this case, with files noticeably bigger, network and servers are saturated enough to make a difference when wisely selecting the replica servers for every file request.



(a) Results for 60 clients

(b) Results for 120 clients

(c) Results for 360 clients

(d) Results for 480 clients

Figure 5.13: *Results of Classic vs Replica Selection Algorithm in Grid environment (FS-MB dataset, simulated Medium Grid platform)*

Figure 5.13 shows a peak in classic round-robin approach for 8 servers. The reason of this peak is that when using 8 servers (compared to using 2, 4, and 6), a dataserver located in *Sophia* is added. *Sophia* is a city located at the south of France, which connects the two 1 Gigabit Ethernet links, *Nantes* and *Reims*, to the rest of Grid5000 (see Figure 4.2). When *Sophia* is added, this causes a bottleneck in the link that connects *Sophia*, and thus *Nantes* and *Reims* too, with the rest of Grid5000 network. *Reims* node was already being used, but when using both *Sophia* and *Reims* the throughput of *Reims* drops significantly, causing a the shown peak for 8 servers.

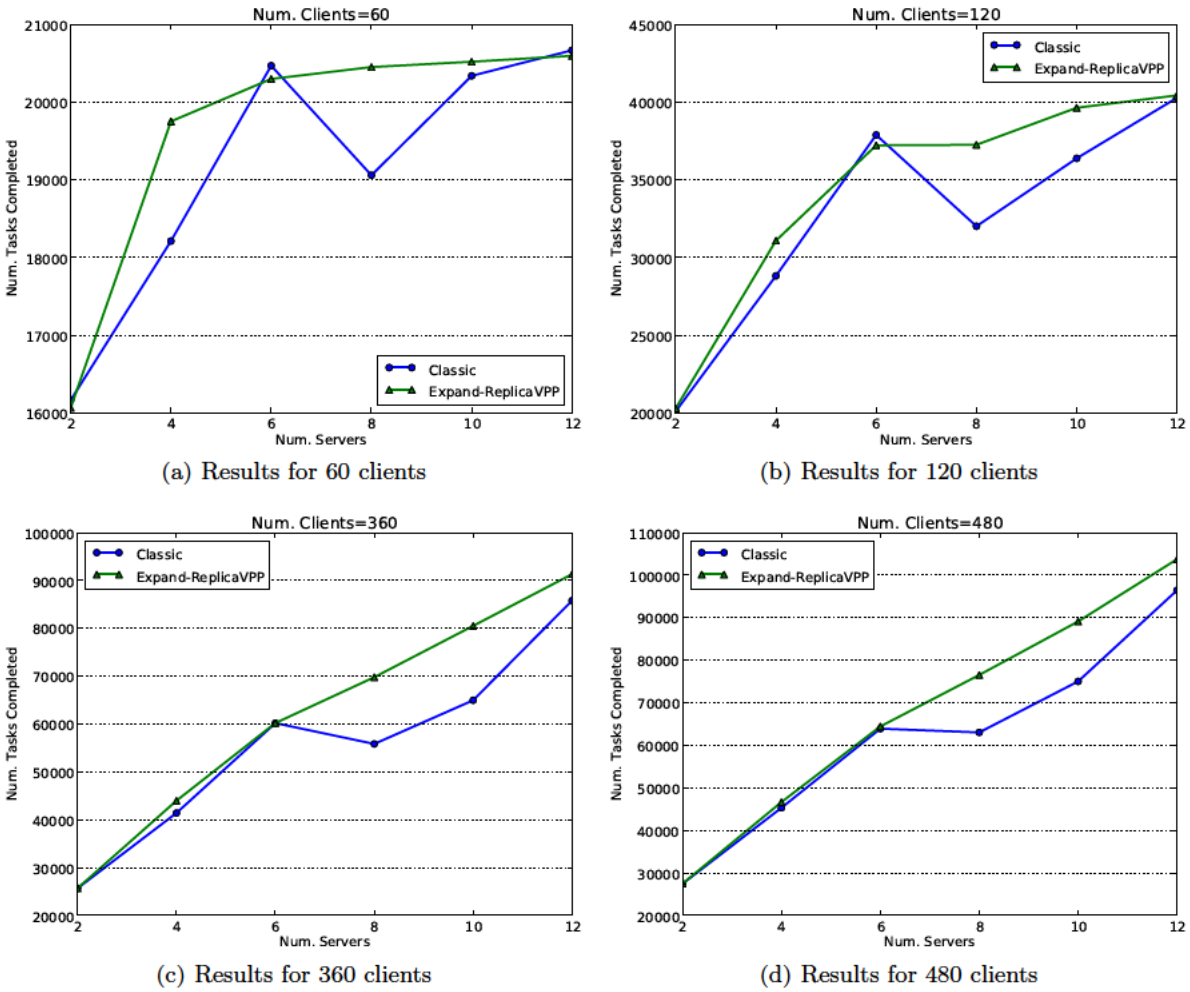
However, the replica selection algorithm proposed in this thesis overcomes this situation, balancing the workload efficiently, and takes into consideration the slow links behind *Sophia*.

5.4.4 Evaluation in volunteer computing environments

For testing the replica selection algorithm an experiment with a duration of one hour was conducted using the simulator. At the end of the hour, the number of tasks completed is counted. Figure 5.14(a) shows results for 60 clients for the FS-MB dataset in volunteer computing environment.

Like in Grid case, if the network or the servers are not saturated enough the algorithm for replica selection shows marginal improvement over classical round-robin algorithm.

However, when the experiment induces high overload over network or servers, results show an improvement when using the replica selection algorithm proposed (see Figure 5.14).



(a) Results for 60 clients

(b) Results for 120 clients

(c) Results for 360 clients

(d) Results for 480 clients

Figure 5.14: *Results of Classic vs Replica Selection Algorithm in volunteer computing environment (FS-MB dataset, simulated Medium Grid platform)*

In this case, with network and servers saturated enough, there is a difference when wisely selecting the replica servers for every file request.

Again, as in Grid case, we observe the same peak for 8 servers. The same reason given in previous section holds here.

5.5 Summary

This chapter has presented the optimizations proposed for large-scale distributed systems, namely, a replica selection algorithm for configuring virtual parallel partitions. The theoretical model has been presented. For the evaluation, a complete simulator of Grid and volunteer computing environments has been designed. Finally, the replica selection algorithm has been evaluated in a simulated Grid and volunteer computing environments using the simulator developed.

Chapter 6

Conclusions and future work

In this thesis we have proposed a generic I/O middleware architecture for large-scale distributed systems, and an algorithm for replica selection to improve data access performance.

The thesis has properly fulfilled all the primary objectives indicated in Section 1.2:

- To propose a **generic I/O middleware architecture** for large-scale distributed systems.
- To design a **replica selection algorithm** for configuring access virtual parallel partitions for large-scale distributed systems.

Our generic parallel and remote I/O middleware architecture, along with an algorithm for replica selection has accomplished a number of benefits:

- **Generic architecture**. The architecture proposed targets large-scale distributed systems. Instead of targeting cluster and supercomputer architectures, like is typically done in high-performance environments, this thesis proposed a generic architecture based on open-source software, encompassing large-scale distributed systems.
- **Portability**. Portability is achieved by using well known technologies like HTTP and GridFTP, de-facto transfer standards in distributed architectures.
- **Scalability**. This architecture has proven to be scalable to systems with thousands of nodes through simulation. The novelty of our approach consists in addressing scalability by using multiple replicas of the data in parallel, and an algorithm for selecting the best set of replicas for every single request.
- **High-throughput**. The application offered high-throughput parallel and remote I/O. High throughput is obtained from parallel and remote access to independent storage resources, a tight integration between the applications and the middleware, and overlap of computation, communication, and I/O.

- **High resources utilization**. The conjunction of the middleware and a novel algorithm for replica selection achieves a high utilization of available resources such as storage and networks.
- **Transparency and simplicity of use**. This architecture can be used transparently by the user. Optionally, the user may chose different parallel I/O optimizations in a straightforward way.

The results obtained in thesis allows to affirm that these objectives have been been fulfilled: a parallel and remote file system, called *Expand*, has been proposed, and adapted, and in conjunction with an algorithm for replica selection, to provide scalability and high-performance in large-scale distributed systems.

The rest of this chapter is organized as follows. We start by describing the contributions of this thesis. Then, we enumerate the publications obtained, and finally we propose new lines of research arising from this thesis.

6.1 Contributions

This thesis presents contributions to the study, design and improvement of data access to large-scale distributed systems. It makes two main contributions to these kinds of environments:

- The proposal of a parallel and remote I/O system, originally designed for clusters and supercomputers, based on standard I/O services, and its adaptation to provide it with well-known and standard protocols and services in distributed systems, and with optimizations needed to operate efficiently in environments with high latencies or low throughputs, that provides compatibility and adaptability in large-scale distributed systems.
- The proposal of an algorithm for replica selection to be used in parallel and remote I/O systems, that selects an optimized subset of replicas for any given request, balancing I/O workload generated by user applications among all the servers.

These two approaches provide new opportunities in the implementation of I/O systems and services for large-scale distributed systems.

This thesis makes the following contributions:

- **Definition of a generic I/O middleware architecture for large-scale distributed systems**. An existing parallel file system, called *Expand*, based on standard I/O services, and that provides remote access has been proposed as the base architecture for large-scale distributed systems. *Expand* has been adapted to provide it with two main capabilities needed for large-scale distributed systems: support for well-known and standard protocols, and low-level optimizations to provide high-performance in remote access to high-latency or low-throughput servers.
- **Evaluation of remote I/O compared to traditional execution of applications**. A comparative study of different ways of execution of applications in distributed environments has been carried out. The main objective of the study has been comparing the traditional

execution of applications in distributed systems, and the proposed scheme of parallel and remote I/O. For this purpose, several types of workloads, and platform environments (Grids and volunteer computing environments) have been tested. The results show that parallel and remote I/O in distributed systems provides better performance than the traditional execution of applications.

- **Definition of an exhaustive benchmarking methodology**. During the course of this work, the necessity of an exhaustive benchmark arose. In order to evaluate the proposals of this thesis, an exhaustive methodology of benchmarking was designed: Balanced Benchmark. This methodology defines balance levels for servers' workload, and different client workloads that generate the different balance levels in the servers. This is a rigorous methodology of benchmarking distributed systems that covers most of the cases of client workloads.

- **Integration of a generic I/O middleware architecture for large-scale distributed systems into BOINC**. In order to evaluate the generic I/O middleware for large-scale distributed systems proposed in this thesis, we have integrated the *Expand* parallel file system into the BOINC software, so that the use of this file system is completely transparent to applications running in BOINC.

- **Design and implementation of a complete simulator for clusters, Grids, Desktop Grids, and volunteer computing platforms**. Performing tests for large-scale scenarios requires access to big platforms, and in case access to those platforms is granted experiments might take long time. To ease the task of evaluating the proposed solutions a complete simulator of parallel and remote distributed I/O has been developed.

- **Proposal of a replica selection algorithm for large-scale distributed systems**. In order to select an efficient subset of replica servers, an algorithm for replica selection has been proposed. On the one hand, this algorithm for replica selection takes into account the size of the requests. On the other hand, the performance of data servers, measured in terms of latency and throughput. The result is that for a given size request, and considering the characteristics of the data servers, an optimized subset of replicas is chosen for configuring virtual parallel partitions to do parallel and remote I/O, while keeping a balance on servers' workloads.

6.2 Thesis results

The principal contributions of the thesis have been published in diverse papers in international conferences and journals. We enumerate the publications classified in four groups: articles in journals, posters, international, and national conferences.

- Journals

  – *SkyCDS: A resilient content delivery service based on diversified cloud storage*
  Journal: Simulation Modelling Practice and Theory, vol. 54, pp. 64–85
  Year: 2015
  ISSN: 1569-190X
  DOI: 10.1016/j.simpat.2015.03.006

- *Expanding the volunteer computing scenario: A novel approach to use parallel applications on volunteer computing*
  Journal: Future Generation Computer Systems, vol. 28, no. 6, pp. 881–889
  Year: 2012
  ISSN: 0167-739X
  DOI: 10.1016/j.future.2011.04.004

- *A novel methodology for the monitoring of the agricultural production process based on wireless sensor networks*
  Journal: Computers and Electronics in Agriculture, vol. 76, no. 2, pp. 252–265
  Year: 2011
  ISSN: 0168-1699
  DOI: 10.1016/j.compag.2011.02.004

■ International conferences

- *Improving MPI Applications with a New MPI_Info and the Use of the Memoization*
  Conference: 20th European MPI Users' Group Meeting (EuroMPI '13), September 15-18, 2013, Madrid, Spain
  Year: 2013
  ISBN: 978-1-4503-1903-4
  DOI: 10.1145/2488551.2488554

- *Fault-tolerant middleware based on multistream pipeline for private storage services*
  Conference: 7th International Conference for Internet Technology and Secured Transactions (ICITST 2012), December 10-12, 2012, London, UK
  Year: 2012
  ISBN: 978-1-4673-5325-0
  URL: ieeexplore

- *Virtual I/O Forwarding for Cloud-based HPC Applications*
  Conference: 2012 10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2012), July 10-13, 2012, Leganés, Madrid, Spain
  Year: 2012
  ISBN: 978-1-4673-1631-6
  DOI: 10.1109/ISPA.2012.139

- *An Efficient Deployment Strategy for Large Sets of Virtual Appliances*
  Conference: 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 09), July 13-16, 2009, Las Vegas, Nevada, USA
  Year: 2009
  DBLP key: conf/pdpta/RodriguezCNBCG09

- *Resource selection for fast large-scale Virtual Appliances Propagation*
  Conference: 14th IEEE Symposium on Computers and Communications (ISCC 2009), July 5-8, 2009, Sousse, Tunisia
  Year: 2009
  ISSN: 1530-1346
  DOI: 10.1109/ISCC.2009.5202254

- *Architecture for improving data transfers in grid using the Expand parallel file system*
  Conference: 3rd Iberian Grid Infrastructure Conference (IBERGRID 2009), May 20-22, 2009, Valencia, Spain
  Year: 2009
  ISBN: 978-84-9745-406-3
  URL: Ibergrid 2009 Programme
- *Comparing Grid Data Transfer Technologies in the Expand Parallel File System*
  Conference: 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), February 13-15, 2008, Toulouse, France
  Year: 2008
  ISSN: 1066-6192
  DOI: 10.1109/PDP.2008.51

■ Posters

- *Improving the performance of the BOINC volunteer computing platform using the Expand parallel file system*
  Conference: Fifth IEEE International Conference on e-Science (e-Science 09), December 9-11, 2009, Oxford, UK
  Year: 2009

■ National conferences

- *Mejora del entorno de computación voluntaria BOINC usando el sistema de ficheros paralelo Expand*
  Conference: XXI Jornadas de Paralelismo, September, 7-10, 2010, Valencia, Spain
  Year: 2010

- *Descripción de una nueva arquitectura de E/S para grandes clusters*
  Conference: XIX Jornadas de Paralelismo, September 17-19, 2008, Castellón, Spain
  Year: 2008
  ISBN: 978-84-8021-6

- *Un nuevo enfoque para implementaciones MPI en entornos Grid*
  Conference: XIX Jornadas de Paralelismo, September 17-19, 2008, Castellón, Spain
  Year: 2008
  ISBN: 978-84-8021-6

- *Adaptación del sistema de ficheros paralelo Expand a entornos Grid*
  Conference: XVIII Jornadas de Paralelismo, September 11-14, 2007, Zaragoza, Spain
  Year: 2007
  ISBN: 84-9732-593-6

Other achievements in this thesis include research stays, and research grants:

- Research stays
  - *Laboratoire d'Informatique de Grenoble*
    Institution: Inria, Grenoble Rhône-Alpes, France
    Hosted by: Dr. Derrick Kondo
    Date: July-October 2011
    Duration: 3 months

- Research grants
  - *Programa propio de investigación, ayudas de movilidad 2011*
    Institution: Universidad Carlos III de Madrid
    Funds: 2,100€

6.3 Future work

There are several lines of research arising from this work which could be pursued:

- **Apply these ideas to different environments, like virtualized and Cloud environments**. Many researchers agree that in the future companies will rely less on their own infrastructures and more on remote clouds. An interesting line of research is to study parallel and remote I/O techniques in these environments to provide higher I/O performance.

- **Study of heterogeneous distributed partitions**. In this work we have assumed that the distributed partitions are homogeneous in terms of the protocols used to serve the data. We propose to study the effect of having distributed partitions on the Internet whose data servers use different protocols to serve the data, and how this heterogeneity affects overall performance.

- **Hybrid remote-memory-disk system for large-scale distributed systems**. This thesis has studied traditional execution of applications, and parallel and remote access to data. However, it is interesting to study a hybrid remote-memory-disk system for large-scale distributed systems. An scalable storage architecture, based on intermediary storage nodes, has already been proposed in other works for clusters and supercomputers. We propose to extend this approach to large-scale distributed systems, to define a hybrid remote-memory-disk system, which makes use local disks for caching data, while providing the data to the application as soon as they come, as in remote access, and completely transparent to the user, but for large-scale distributed systems.

- **Integration of *Expand* in Big Data environments**. Apache™ Hadoop® is the reference technology in the emerging Big Data ecosystem. Apache™ Hadoop® (on top of HDFS™) is a massively distributed data environment, that offers a MapReduce programming interface to ease the development of distributed applications that need to process massive amounts of data. We propose to integrate *Expand* in Apache™ Hadoop®, and to study how a MapReduce application can benefit from this architecture.

- **Design a Spark-like backend into *Expand***. Among all technologies around the Big Data ecosystem, the most promising technology nowadays is, probably, Apache Spark™. Apache Spark™ is a fast and general-purpose cluster computing system, which loads the dataset required by an application into memory, so that all operations over the dataset are carried out in memory. Spark's operators spill data to disk if it does not fit in memory. Likewise, cached datasets that do not fit in memory are either spilled to disk or recomputed on the fly when needed. We propose to design an Spark-like backend into *Expand*, so that unmodified single node (though possibly multicore) applications can benefit from parallel and remote access to a distributed dataset cached in memory.

- **Define new architectures in *Expand* that decouple data and metadata**. Currently, *Expand* architecture is strongly dependent on coupling data and metadata. We propose to define new architectures that decouple data and metadata, through a fault-tolerant and high performance metadata system using Apache Zookeeper™.

- **Implementation and study of the replica selection algorithm in real environments**. The algorithm for replica selection proposed in this thesis is only a sketch, and it has only been tested through simulation. Clearly, this algorithm for replica selection should be implemented into *Expand* and tested in real environments in order to evaluate its feasibility as an effective replica selection algorithm. But this is a field of study by itself. We propose to study this question more in depth, for all the possible situations, use cases, and implications around the problem of selecting the optimal subset of replicas for configuring virtual parallel partitions in large-scale distributed systems.

- **Make use of a replica location service**. Currently, the replica selection algorithm implementation is based on a static list of replica servers. However, during the life-time of an evolving environment, replicas are often created and removed. We propose to design or make use of a low latency replica location system, so that the algorithm for replica selection can select better replicas when available, or eliminate dead replicas when they are no longer available.

- **Self-adjusting replica selection algorithm**. Currently, the replica selection algorithm implementation selects a subset of replicas based on weights functions that are configured statically. However, the optimal parameters for these functions strongly depend on the platform. We propose to design a self-adjusting mechanism, so that, the weights of the functions can be tuned during the life-time of the system, looking for the optimal configuration in every moment.

Appendix A

Architecture, design, and implementation of *Expand*

This chapter describes the *Expand* parallel file system. The architecture of this file system is used as a reference for the model of parallel file system for large-scale distributed systems proposed in this thesis.

The main motivation in the design of *Expand* was to build a parallel file system for heterogeneous clusters using standard servers. To this aim, the Computer Architecture, and Technology group of Universidad Carlos III de Madrid designed, and implemented a parallel file system using NFS servers. The firsts prototypes have been been described, and evaluated in the given bibliography [García Carballeira et al. (2002), Calderón et al. (2002a,b), García Carballeira et al. (2003a,b,c), Calderón et al. (2003), Sánchez García (2003), Bergua Guerra (2006), Bergua et al. (2007)]. And later, more in depth in [Calderón (2005)], and more formally in [Sánchez García (2009)].

As *Expand* will be used as a base parallel file system to implement the ideas and proposals of this thesis, this chapter presents previous works on *Expand*, including the reference architecture, the parallel file model used, data and metadata distribution, naming, parallel access, and user interfaces.

A.1 Introduction

Heterogeneous clusters consist of compute nodes running different operating systems, and software, which limits the amount of file systems suitable for such systems. This set of suitable file systems comprises NFS or CIFS, which are standard in many computing environments. This difficulty to incorporate parallel file systems to these kinds of environments prevents the use of techniques for parallel access to data, and reduces the performance of storage systems.

The *Expand* parallel file system has been designed to face these problems, which uses standard

data servers like NFS, to form parallel storage systems.

In the following sections the characteristics of this parallel file system will be detailed.

A.2 General overview of the architecture of *Expand*

In the generic architecture of a parallel-distributed I/O system different functional levels can be distinguished [Eckardt (1990)]:

- Process Node (PN). Perform I/O requests. Also known as client.
- I/O Node (ION). Carries out I/O operations over files and logic devices. Also known as server.
- Controller. Transfers physical blocks from/to physical devices.
- Device. Store data in physical blocks.

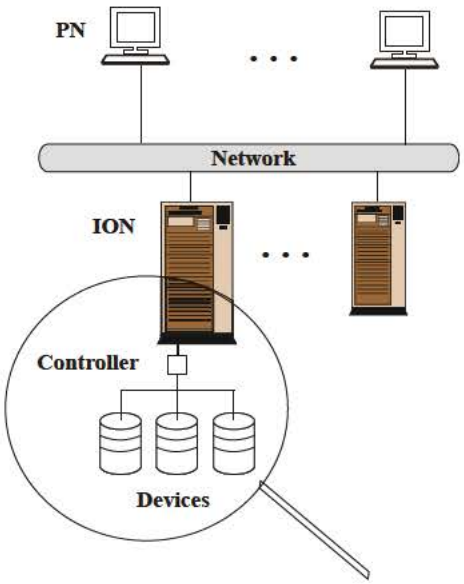


Figure A.1: *Generic architecture of a parallel file system*

The traditional architecture of a parallel file system can be described based on the above definition and generically as shown in Figure A.1. This architecture has a set of clients (which are process nodes) and a set of servers interconnected by a network. These servers are input and output nodes which the different devices connect to across different controllers.

The servers work together to provide clients with the view of a single storage system [Hwang et al. (1999)] that gathers the storage capacity provided by each server. Clients demand, through the set of servers, storage and retrieval operations of data in the storage space offered by servers. Unlike servers, a client is not, a priori, aware of the existence of the others. Each client operates independently, even if they are different processes of the same parallel application.

The servers, in addition to managing the storage system offered to clients, have to ensure data consistency when multiple clients access the same piece of stored data. Examples of parallel file

systems are as follows: ParFiSys [Carretero et al. (1996), Pérez et al. (1997)], Galley [Nieuwejaar and Kotz (1996a,b, 1997)], Vesta [Corbett et al. (1993)], PIOUS [Moyer and Sunderam (1994)], PVFS [Carns et al. (2000)], Armada [Oldfield and Kotz (2001)], etc.

The proposed architecture for the *Expand* parallel file system is based on a different approach to the vast majority of parallel file systems:

- To offer a parallel file system that allows its integration in heterogeneous systems it is possible to use standard data servers.
- Since there is no specific set of servers that are intermediaries between clients and existing servers, clients have to perform some of the tasks required to provide a parallel file service.

Figure A.2 shows the architecture of *Expand*. The *Expand* file system is designed according to the client/server model, where the client side of the file system is responsible for receiving requests from processes through different types of interfaces, such as *MPI-IO* or *POSIX*, and the server side is formed by any standard networked file server.

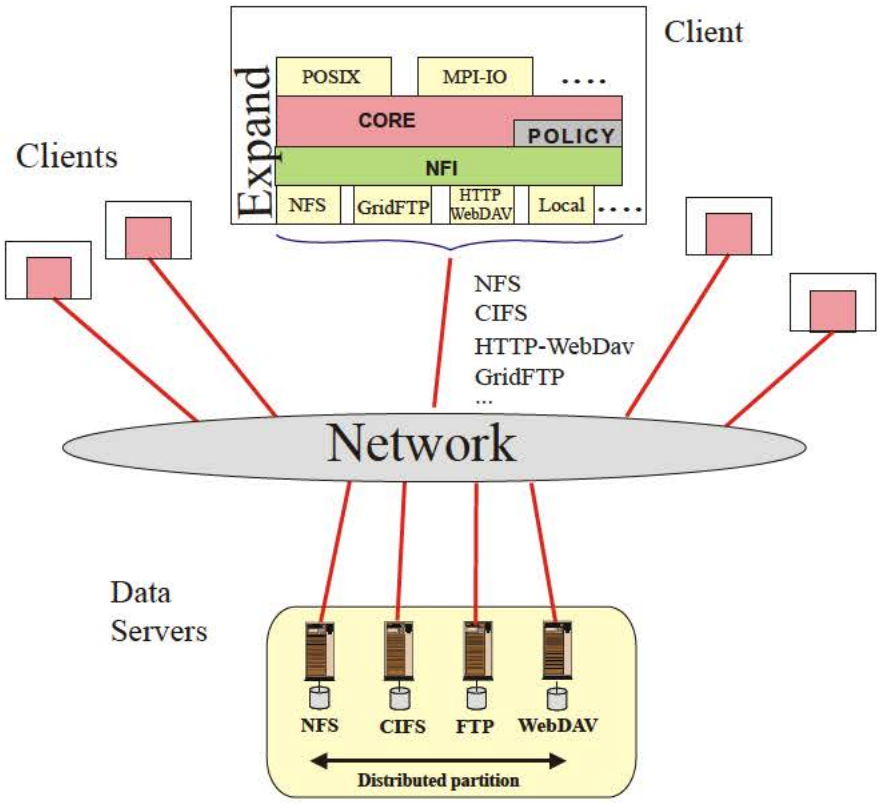


Figure A.2: *Architecture of Expand*

The *Expand* client is responsible to contact the servers using the protocols associated with them. For this, the client is designed as shown in Figure A.2, divided into four layers.

The top layer is responsible for providing various access interfaces to parallel applications. The layer named *core* is responsible for defining the basic algorithms used for data access. Inside this layer is located the module called *policy*, that is responsible for defining what are the actions

to take in the operations of metadata location, selection of servers involved in every operation, etc. Both *core* and *policy* use the services of the layer called *NFI*, which stands for *Network File Interface*. This layer provides an interface to the basic operations of a file system. The lower layer is responsible for implementing the interface provided in the *NFI* layer for the different access protocols to a file system.

The data of a file are scattered by *Expand* through all servers using blocks of a certain size as distribution unit. The processes of a parallel application are the clients that use the *Expand* library to access to a distributed partition.

Expand provides an interface based on POSIX calls. However, this interface is not suitable for parallel applications that use stridden patterns with small access sizes [Nieuwejaar and Kotz (1996a)]. For parallel applications *Expand* has been integrated within *ROMIO* [Thakur et al. (1999a,c)] which allows it to access to the *MPI-IO* interface.

Using the outlined approach for the design of *Expand* we have the following advantages:

- No changes are required on the servers (NFS, WebDAV, FTP, etc.) to benefit from *Expand*. All aspects of the operations of *Expand* are deployed in the client.
- *Expand* is independent of the operating system used on clients. All operations are implemented using standard protocols.
- It allows the use of servers running on different architectures and operating systems, since the use of standard protocols hides these differences.
- Building a file system is greatly simplified because all operations are implemented in clients. This approach is completely different to that used in many of the current parallel file systems such as CFS [Pierce (1989)], Vesta [Corbett et al. (1993)], HFS [Krieger (1994)], PIOUS [Moyer and Sunderam (1994)], Scotch [Gibson (1995)], Galley [Nieuwejaar and Kotz (1996a,b, 1997)], ParFiSys [Carretero et al. (1997)] or PVFS [Carns et al. (2000)].
- The configuration of the system is much simpler as standard servers, such as NFS, are very familiar to users. The server only needs to export the appropriate directories, and clients only need a small configuration file detailing how is the distributed partition.

Other systems use a standard server as a basis for their work, similar to *Expand*. For example, Bigfoot-NFS [Kim et al. (1994)] also combines multiple NFS servers. However, this system uses a file as distribution unit and, therefore, all the data of a file reside on a single server. Although the files in the same directory can be distributed across multiple machines, this system does not allow parallel access to the same file.

Another similar system is the *Slice* file system [Chase et al. (2000)]. *Slice* is a storage system for high speed networks that uses a microproxy (called μproxy) as a packet filter to virtualize an NFS server, so that it presents a unified and shared file volume to NFS clients. This system uses the μproxy to distribute file service requests among aggregated servers, providing compatibility with file systems on clients. However, the μproxy can become a bottleneck affecting the overall system scalability. This is a serious disadvantage compared to *Expand*.

Another similar solution is developed by *Avaki* [Avaki Corporation (2005)]. *Avaki* offers a commercial solution for providing data access in Grids (see Figure A.3). It is based on Legion [White et al. (2001)], developed by the University of Virginia, and its aim is the management of

Data Grids such that different resources are available under a common namespace. It offers an NFS interface to clients.

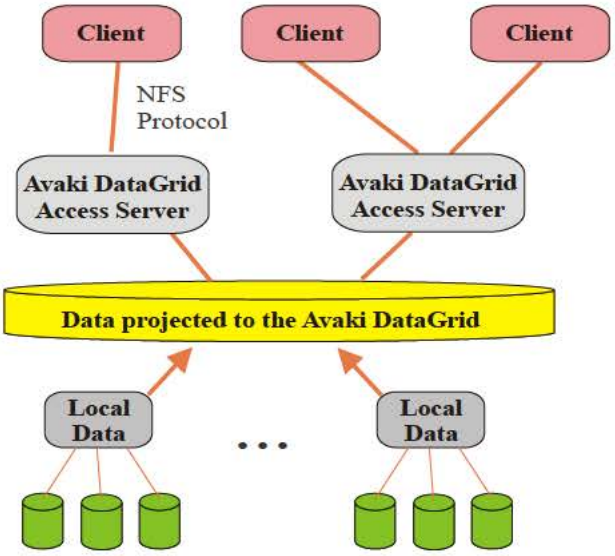


Figure A.3: *Architecture of Avaki*

In case of a fail [Berman et al. (March 2003)] [Foster et al. (2003)], jobs and needed files are migrated. If a machine needs to be disconnected from the *Grid* for some reason, administrators can migrate files to another machine. The requests to these files can be redirected to the new location automatically. This system offers fault-tolerance support in the access to data based on replication, which presents the associated disadvantages.

A.3 Model of distributed partition

Expand uses the storage that servers can offer to store files. A storage unit is defined by a server that provides storage, a protocol that is used to communicate with the server, and an access point on that server.

Therefore, a storage unit (referred to as SU) can be defined as:

$$SU \equiv \{ Server,\ Protocol,\ Remote\ directory \} \tag{A.1}$$

The remote directory is a directory exported by the server, should be an NFS server, a shared folder for CIFS, etc. The protocol that operates a storage unit offers a number of services that are the functionality given in the *NFI* layer of *Expand*.

In particular, the functionality provided by *NFI* allows:

- Connect/disconnect to/from a server.
- Create/remove/open/close a file.
- Read/write data from/to a file.

- Create/remove/open/close a directory.
- Read entries of a directory.
- Read/modify attributes of system's object (file or directory).
- Check file system status, for example indicating the free blocks.

Finally, the system identifies the server that provides the storage unit to be used.

A property of a storage unit is that it allows to store files, and each file has a unique name within that storage unit. A storage unit thus contains a set of files and directories that are managed by the local file system of the server that provides such storage unit. This involves two levels of metadata management, one in the storage unit level, and other in the parallel file system level. *Expand* only manages the latter metadata.

In a single server there can be multiple storage units. This allows you to add more storage units as needed, offering more flexibility to users. However, the use of multiple storage units in the same server can result in a lower performance of the *Expand* parallel file system.

Expand combines multiple storage units to form a parallel and distributed partition (see Figure A.4). Therefore, a parallel and distributed partition is defined as:

$$PDP \equiv \{ \bigcup_{i=0}^{n} SU_i, M_{pdp} \} \tag{A.2}$$

That is, as the union of several storage units, along with some metadata about the partition, called M_{pdp}. The main attribute of these data is the default stride size of files among different storage units.

A.3.1 Configuration file

Expand partitions are defined using a small configuration file in XML format [World Wide Web Consortium (W3C) (2008)], with the following structure:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xpn_conf>
    <partition name="<partition1>" type="<options>" bsize="<block_size>">
        <data_node id="id1" url="<protocol>://<server>/path/"/>
         ...
    </partition>
     ...
    <partition name="<partitionN>" type="<options>" bsize="<block_size>">
        <data_node id="id1" url="<protocol>://<server>/path/"/>
         ...
    </partition>
</xpn_conf>
```

Where the label *partition* allows the definition of the structure of a storage partition. This partition is characterized by four basic elements:

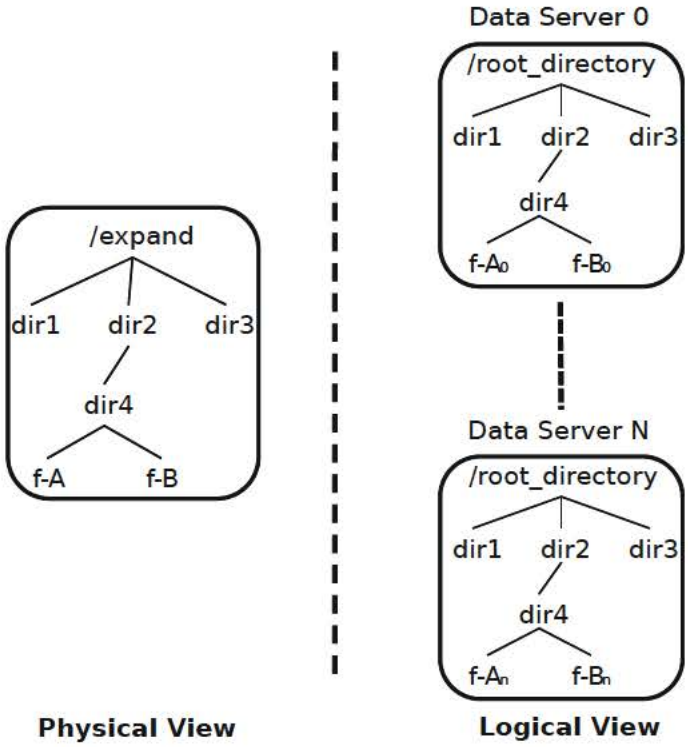


Figure A.4: *File and directory structure in Expand*

- The name of the partition (N_P), which uniquely identifies the partition declared.
- A partition type (T_P), used to define different types of policies to be used in the distribution of the data, which determines the distribution function (f).
- A block size (S_P), used to distribute the data of the files.
- And finally, a set of storage units used in the partition (SU_P).

$$P = \{N_P, T_P, S_P, SU_P = \bigcup_{i=1}^{n} SU_i\} \tag{A.3}$$

For each partition, we define each one of the servers used to form the partition. The servers are defined by a URL which is unique within a partition, not being able to use the same URL more than once per partition. The URL is made up of all the elements needed to handle the data server:

- A communication protocol (P_E).
- The address of the server (S_E), as well as a port and user, if needed.
- A directory (D_E), which will store the user data.

$$SU = \{P_E, S_E, D_E\} \tag{A.4}$$

Thus, for example, the following configuration file defines two distributed partitions:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xpn_conf>
    <partition name="xpn1" type="RAID0" bsize="8k">
        <data_node id="su0" url="nfs://node0/export/home1/"/>
        <data_node id="su1" url="nfs://node1/export/home2/"/>
        <data_node id="su2" url="nfs://node2/export/home3/"/>
        <data_node id="su3" url="nfs://node3/home/"/>
    </partition>

    <partition name="xpn2" type="RAID0" bsize="8k">
        <data_node id="su4" url="xio://node5/scratch/"/>
        <data_node id="su5" url="xio://node6/export/scratch/"/>
   </partition>
</xpn_conf>
```

The first uses four storage units (*SU-0*, *SU-1*, *SU-2*, and *SU-3*) with NFS protocol, version 2. The second partition uses two storage units (*SU-4*, and *SU-5* with XIO protocol. In both cases blocks of 8 KB are used as stride unit. The `/xpn1` directory is the root directory for the first partition, and `/xpn2` the root directory for the second. Using these files, the *Expand* file `/xpn1/dir/foo` is projected on the following subfiles, as seen in Figure A.5.

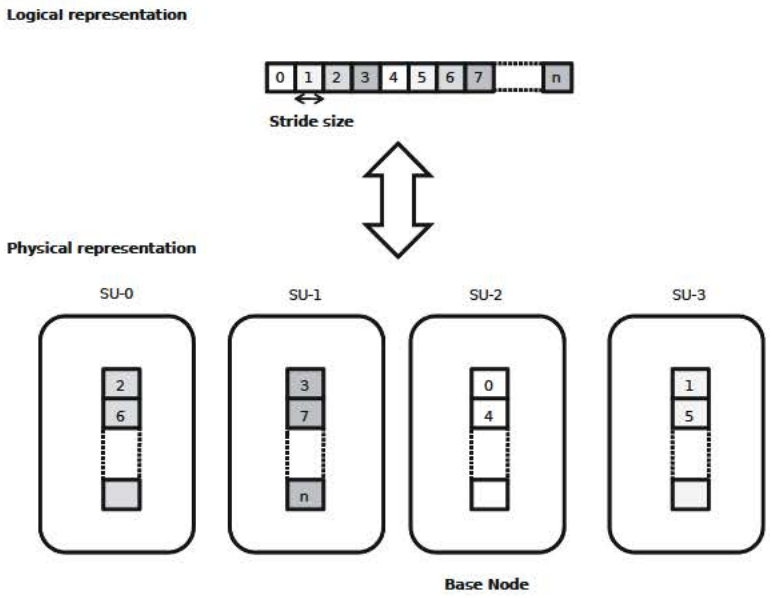


Figure A.5: *Example of data projection*

This scheme allows even having a distributed partition with different protocols, which encourages the use of multiple heterogeneous servers to form distributed partitions transparently to users.

A.4 Model of parallel file

After introducing the concept of distributed partition, we proceed to introduce the concept of parallel file that resides on a distributed partition.

This section analyzes aspects such as the distribution of the data, metadata, directory and file naming.

A.4.1 Definition of parallel file

A file (F) has a projection in *Expand* (F_P) by a function f_e. A parallel file (F_P) is defined as:

$$f_e(F) \rightarrow F_P : F_P \equiv (D, M) \tag{A.5}$$

Where D is the user data distributed in the storage units defined in *Expand*, and M is the set of metadata associated with the parallel file F_P. Data D and metadata M are distributed over several storage units SU, which are grouped into distributed partitions (P) (see Figure A.4), and each SU consists of a data server (SU_D), and an exported directory (SU_E).

In *Expand*, for every file there is a subfile in each storage unit (storage unit file) of the distributed partition.

User data (D) of parallel files (F_P) are distributed through storage units (SU) through subdivision into blocks (B) of equal size. Data blocks (B_i) are represented as the set of tuples of the form (*offset*, *value*), where *offset* uniquely identifies the tuple with the rest of tuples of the file, and *value* is the data to be stored in the tuple (see Definition A.6). The *offset* allows the location of a *value*, forming an ordered sequence of tuples (see Definition A.7).

$$B_i \equiv \{(o_i, v_i)\} \tag{A.6}$$

$$D \equiv \bigcup_{i=i}^{n} B_i \equiv \{(o_1, v_1), ..., (o_n, v_n)\} \tag{A.7}$$

This abstraction is not new, it was used in Linda [Ahuja et al. (1986)] to provide a communication system among cooperating processes using some memory sharing mechanism.

Both the data contained in the parallel file, as well as the data contained in the subfiles stored in storage units are represented along this abstraction.

These data blocks (B) are grouped into subfiles (S) stored in different storage units (SU). A subfile (S_i) consists of one or more sorted data blocks (B_{ij}), where i represents the subfile i, and j the internal order in it. Moreover, $\|B_i\|$ represents the number of blocks stored in S_i. Each storage unit only has one subfile per parallel file.

$$\forall SU_i \in P, \exists S_i : S_i \subset SU_i \tag{A.8}$$

$$S_i \equiv \bigcup_{j=1}^{n} B_{ij} : n \in I\!N \rightarrow \|B_i\| \subset S_i \tag{A.9}$$

$$\forall SU_i, SU_j \in P, i \neq j : SU_i \cap SU_j = \emptyset \tag{A.10}$$

Each subfile S_i stores part of the data of the parallel file, so that the union of the data contained in the subfiles (following a predetermined order) represents the content of the parallel file, ie:

$$Data(S_i) \equiv \bigcup_{j=i_0}^{i_n} \{(o_j, v_j)\} \tag{A.11}$$

$$Data(F_P) \equiv \bigcup_{i=0}^{s} Data(S_i) \tag{A.12}$$

Each subfile is stored in a storage unit. Since each subfile is stored in a storage unit, the server that manages the storage unit will handle the metadata associated with this subfile. These are: stride unit (block size), creation date, modification date, etc.

The metadata contains the distributed partition (P) where the file is stored, and the distribution function f_d that indicates the distribution of the data blocks in the different subfiles stored on their respective storage units.

Each server provides, at least, one storage unit. Storage units are independent of each other, and each one can be managed by a local storage management system in the server.

Parallel file systems access to different storage units in parallel, being each storage unit in a server. A storage unit is provided by a server, but a server can provide multiple storage units. To provide a higher level of parallelism in a distributed partition, and so on all files belonging to it, it is desirable that each storage unit resides on a different server. Thus, there is no server that provides two storage units, to prevent it to be a bottleneck in the access to the data stored in its storage units when done in parallel.

To find out where in the subfile S_i of D is stored the information of a tuple of data (x, v), a distribution function f_d is used. This function is the correspondence between an offset (used to indicate the location of the file data), and a tuple (SUI_i, x_j). In such tuple, SUI_i is the unique identifier of the storage unit SU_i, which the associated subfile is stored in, and x_j is the offset of the user data within the subfile.

$$f_d : I\!N \longrightarrow I\!N \times I\!N \tag{A.13}$$

For all the data of the parallel file (x, y), the function f_d allows to know the subfile, by means of the storage unit identifier SUI_i (which is the unique identifier of the storage unit SU_i), and the location x_j within that subfile whose associated tuple contains the value y.

$$\begin{gathered} \forall\ (x, v) \in D \\ \exists\ S_i \in S \wedge\ \exists (x_j, v) \in S_i \wedge S_i\ stored\ in\ SU_i, \\ f_d(x) = (SUI_i, x_j) \end{gathered} \tag{A.14}$$

For a parallel file, it is desirable that the distribution function provides a distribution of user data with a balanced use of free space, maximum performance, and the best possible reliability.

A.4.2 Data distribution

As was detailed above, each file in *Expand* has several data subfiles (S_i), one for each partition used in each storage unit (SU). S_i is an independent file managed by a server working as a SU, totally transparent to *Expand* clients, and users. The user can create files with different parallel distribution policies in a distributed partition. For example:

- Cyclic distribution. This distribution (f_c), which is shown in Figure A.6, is the most used in most parallel file systems. This distribution system requires a *base* storage unit (SU_b) from which starts the distribution of data blocks.

$$f_c(SU_b, B_{ij}) \rightarrow SU_j : SU_j \in P \tag{A.15}$$

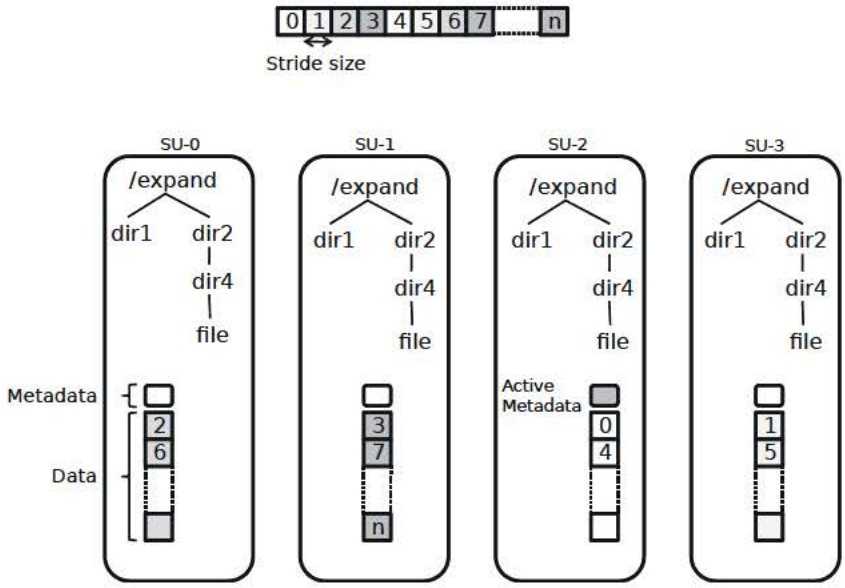


Figure A.6: *Data distribution in Expand*

As shown in Figure A.6, the first block (block with index 0) is stored in the storage unit labeled as *SU-2*, the following in *SU-3*, the following in *SU-0*, the following in *SU-1*, and so on.

- Random distribution. Where a distribution function based on a random pattern (f_a) is used. This scheme creates a seed file that will be used for the generation of pseudorandom numbers. The range of these numbers will be between 1 and the total number of storage units ($\|SU\|$) used in the partition (P). Thus, the block B_i of the file is found in the subfile S_i stored in the SU indicated in the i-th pseudorandom number.

$$f_a(x, B_{ij}) \rightarrow SU_j : SU_j \in P, \quad 1 \leq x \leq \|SU\|, x \in \mathbb{N}, SU \in P \tag{A.16}$$

- User-driven distribution. The user can specify the distribution function of blocks (f_u) in the creation of the file, similarly as done in PVFS [Carns et al. (2000)]. Thus, for example, if the number of SU is 8 and the user indicates the pattern $1-3-4-5$, the file will be distributed cyclically using only these 4 SU.

$$f_u(\bigcup_{i=a}^{b}\{SU_i\}, B_{kj}) \rightarrow SU_k : SU_k \in P \tag{A.17}$$

Other data distribution policies based on the available free space in nodes have been studied [Sánchez García et al. (2004)]. For this study a node management service called *broker* that communicates with the I/O nodes and applications was established. The scheme of the architecture, called *AdExpand*, can be shown in Figure A.7. To avoid bottlenecks or single points of failure, the *broker* service can be found replicated, so regular communications among them are established in order to have a common vision of the system. In the *AdExpand* architecture, *brokers* determine the distribution pattern of the data based on the free space available at the I/O nodes at the time of the creation of a file in *Expand*. The nodes are added or removed from the system through the *brokers*, allowing to provide great flexibility to the system.

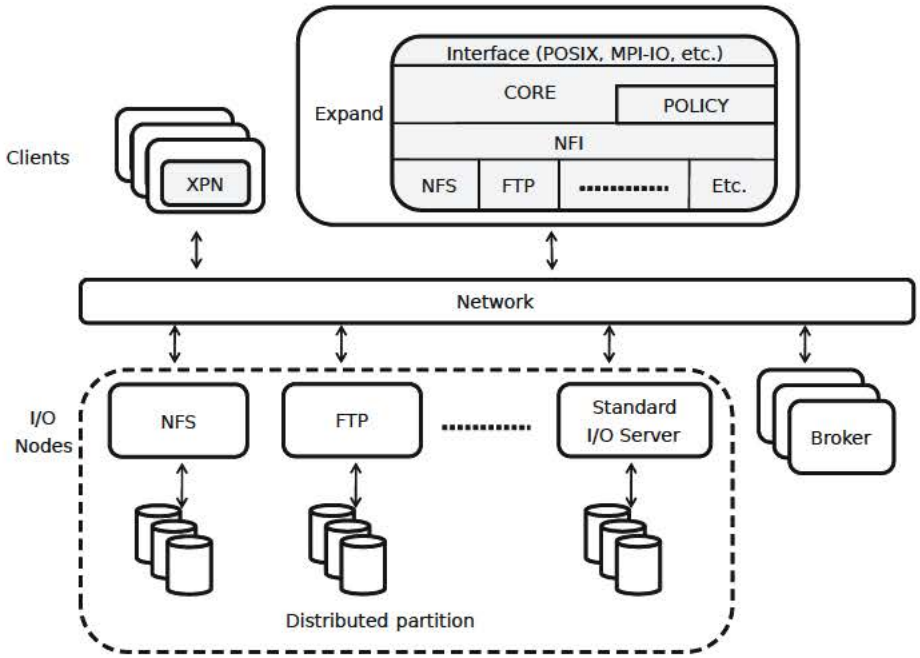


Figure A.7: *AdExpand architecture*

Defined policies based on the free space available in the nodes are:

- NNREP (N servers, no repetition). The client application that aims to create a new file in an *Expand* partition makes a request to a *broker* service, as described above, to request N data servers among all existing servers. The servers returned are those with the highest percentage of free space available. This allows small servers, with smaller capacity, serve less requests, thereby maintaining the system more balanced.

- NREP (N servers, with repetition). The client application performs an action similar to that indicated above, with the difference that a node may appear more than once in the set of nodes returned. The N servers returned by the *broker* are selected as follows: first, the servers with more than 90% of free space are added to the set; next, the servers with more than 75% of free space; and finally, the servers with more than 50% of free space. If there are not any nodes with more than 50% of free space, then the NNREP policy is used. The NREP policy allows a server appear several times in the set of nodes where the data of a file will be distributed. The aim of this policy is that those servers with more free capacity serve more requests.

To distribute the blocks with the user data, it is possible to use the above different distribution policies. *Expand* uses, by default, a cyclic distribution policy (f_c), also known as round-robin or *RAID0*.

User data is divided into blocks of size equal to the stride unit. As shown in the top of Figure A.8, the data are divided into blocks which can be enumerated, so that each block is identified by a positive integer. As shown in the bottom of Figure A.8, each storage unit stores a subfile containing part of the blocks of the user data. In addition, each storage unit stores a small block whose content is the set of metadata of the file.

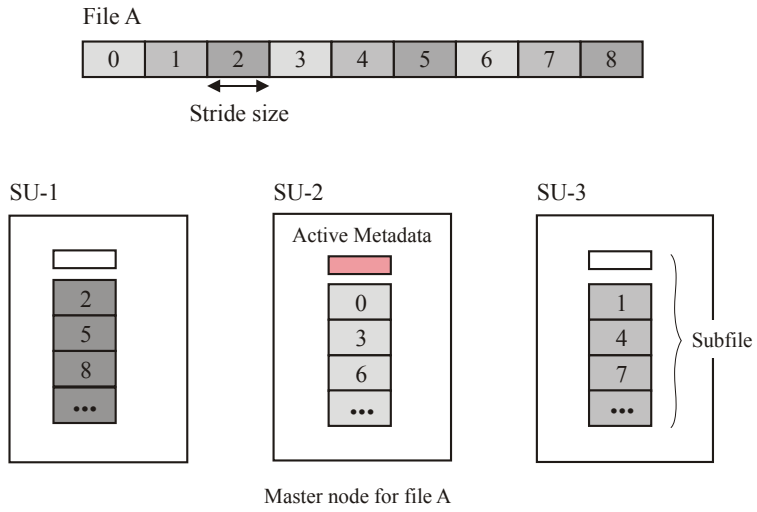


Figure A.8: *Structure of a file in Expand*

Despite reserving space for metadata on all subfiles of each storage unit, only one of the subfiles stores the actual metadata of the file. The idea is to distribute the metadata of different files in a partition among all storage units, to distribute the workload of managing these metadata. The storage unit that stores the metadata of a parallel file is called *master node*

Despite all files can start in the same storage unit, the various files in a partition start in different storage units. The idea, again, is to distribute the workload among the servers that manage the associated storage units. Therefore, the distribution of blocks of each file begins in a different storage unit. That where the first data block stands, and from which the rest of the blocks are distributed cyclically is called *base node*. This *base node* is part of the metadata of the parallel file. Initially (at the time of creation of the file) the *base node* and the *master node* for a file match.

Later sections detail the metadata management, and how to determine the *master node* of a file, as well as the evolution that can take the *base node* and the *master node* over the life of a file parallel.

A.4.3 Metadata

In *Expand* it is possible to distinguish two types of metadata depending on the level in which the management is performed:

- Low-level metadata (M_E), belonging to the subfiles generated by *Expand* in each storage unit. They are managed by each server independently and transparently to *Expand*. Such metadata includes, among others, the file owner, access permissions, and information on the location of the blocks used to store the various subfiles in the storage devices.
- High-level metadata (M_P), involving the *Expand* file. They are managed directly by *Expand*. They include, among other, the following information: the size of the stride unit for each file (the same partition can store files with different stride units), the *base* storage unit (which identifies the storage unit that stores the first block of the file, and the distribution pattern to use), etc.

Thus, the metadata (M) of a parallel file (F_P) comprises:

$$M = \{M_P, \bigcup_{i=1}^{n} M_{Ei}\} : M_{Ei} \in E_i \tag{A.18}$$

For the design of the high-level metadata several options have been considered:

- Metadata management service: system used by PVFS [Carns et al. (2000)] or GPFS [Schmuck and Haskin (2002)], where an external service manages the metadata, which can cause a bottleneck in the system. Other file systems like PVFS2 [Argonne National Laboratory (2011)] have decentralized metadata management to avoid such problems. Moreover, having an exclusive service for handling metadata can increase the complexity of the system.
- Header metadata along with data (B_M): in this case you can access metadata in the same way that data, storing them as a data block in a subfile S_m, which facilitates the implementation of the file system.

$$S_M \in SU_M : \forall\{B_i\} \cup B_M \in S_M \tag{A.19}$$

This metadata can be replicated in each of the files or distributed by some kind of distribution function. In addition, due to the fact that the metadata are included along with the data, you can quickly access them, since the local file system of the storage server can cache them in memory. There are two problems in this management mechanism, first a method for maintaining the consistency of metadata has to be established, and secondly, by including information within subfiles, metadata can not grow, making it difficult to incorporate new information in the metadata.

- Subfile with the metadata content (S_M): like the previous case, the metadata is stored as data, which facilitates its handling by the data server. Furthermore, this solution allows separating data from metadata, a situation that was not possible in the previous scheme. This scheme favors a greater easiness in the creation and management of replication and fault tolerance schemes. Moreover, it allows the metadata to grow unlimited, facilitating the inclusion of more information in the metadata. In contrast, as in the previous case, mechanisms to ensure the consistency of metadata have to be established.

$$F_P = \bigcup_{i=1}^{\|SU\|} \{S_i\} \cup S_M \in P \tag{A.20}$$

- Precomputed metadata: this is the last solution proposed, and is the most agile of those studied so far for managing metadata. This is because physically storing metadata on a device is not needed, as they are generated by the system configuration and by the information of each of the subfiles that represent the data of the parallel file. For example, the size of a parallel file can be obtained from the sum of each of the sizes of the different subfiles which form the parallel file and are defined in the configuration file (see Equation A.21). This metadata management mechanism has some shortcomings, such as the impossibility of maintaining individualized information of files, or the difficulty generated in some of the file management operations, such as renaming or moving.

$$f_{size}(file) = \sum_{S_i \in F_P(file)} f_{size}(S_i) \tag{A.21}$$

Expand does not use any metadata manager, as is the case of PVFS [Carns et al. (2000)]. The two main advantages of this strategy are:

- It eliminates a single point of failure, because if the metadata manager fails, the access to all files on the partition fails.
- It eliminates the bottleneck caused by the metadata manager, since it spreads the load associated with managing the metadata across servers.

The metadata management mechanism based on the use of an external subfile has been chosen for *Expand*. This external subfile that contains the metadata of the parallel file is stored in one of the storage units of the distributed partition (see Figure A.4). For the management of the data, this subfile incorporates precomputed metadata, such as, creation, modification, and access

times, or file size, using for this purpose algorithms designed for obtaining them from the low-level metadata information of each data subfile.

The metadata subfile of an *Expand* file is stored in one of the storage units of the partition called *master node*. This node may be different from the *base node*, where the first block of the file is stored. To simplify the naming process and reduce potential bottlenecks, *Expand* does not use a metadata manager. Instead, the management of the metadata is distributed among the data servers of the partition.

The metadata of a file stores the following information:

- Size of stride unit.
- File type: with or without fault tolerance support.
- Base node identifier, i.e. the storage unit that stores the first data block.
- Distribution pattern. In the current implementation this pattern is cyclical.

Each server is, also, responsible for managing the metadata associated with the subfiles that are stored in it.

A.4.4 Naming

The files that belong to a distributed partition are distributed in the subfiles of each storage unit, as shown in Figure A.9.

As shown in this figure, for each storage unit a subfile with the same name as the parallel file is created. Each of the subfiles resides in the same hierarchy of directories as the parallel file, since such hierarchy is replicated in all storage units, and each of the subfiles contains a part of the data blocks of the parallel file. Specifically, data blocks are distributed cyclically among the sequence of storage units that form the distributed partition.

Therefore, the file `/xpn1/Dir2/Dir4/input.dat`, which belongs to the distributed partition `xpn1` in *Expand*, is projected to the following subfiles:

```
/export1/Dir2/Dir4/input.dat
/export2/Dir2/Dir4/input.dat
/export3/Dir2/Dir4/input.dat
```

For the local storage system of servers `server1`, `server2`, and `server3` respectively. These correspond, in turn, with the storage units `SU-1`, `SU-2`, and `SU-3` respectively. The name of a file in *Expand* has the following format:

```
/<distributed partition name>/<directories/<file name>
```

Where the distributed partition name uniquely identifies the partition, listed in the configuration file that defines the various existing distributed partitions. The directories form the *path* to reach the file.

The filename is very important to locate the master node, which identifies the storage unit whose subfile stores the metadata of the parallel file. The following section details how to perform that location.

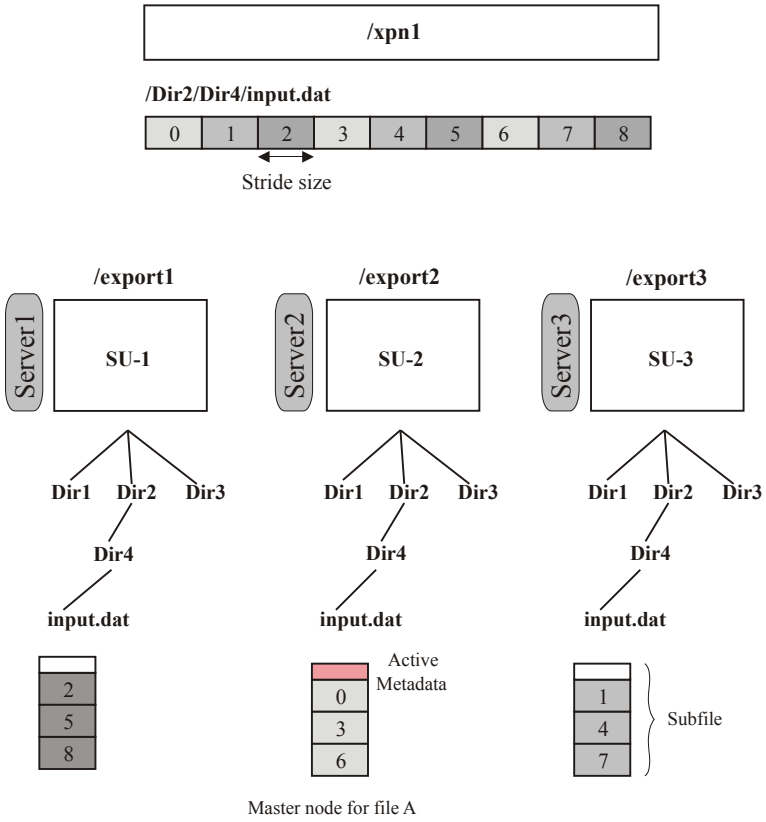


Figure A.9: *The structure of a file in Expand*

A.4.5 Location of master node

To get the metadata of a file it is necessary to access the header of a subfile that resides in one of the SU of the distributed partition called *master* (SU_M). To know which SU stores this subfile, a method similar to that employed in the Vesta parallel file system [Corbett et al. (1993)] is used, which consists of a localization function f_M:

$$f_M(filename, \|SU\|) = SU_i, \quad 1 \leq i \leq \|SU\|, i \in \mathbb{N}, SU \in P \tag{A.22}$$

In the current implementation, the distribution function (f_M) is:

$$f_M(filename) = \left(\sum_{i=1}^{strlen(filename)} filename[i] \right) mod \ (\|SU\|) \tag{A.23}$$

That is, the result of adding the value of the *ASCII* table for each of the characters that form the name of the file, and performing the modulus operation with the number of storage units.

Using this solution has two advantages:

- Facilitates the rapid location of metadata.
- Offers a good distribution of nodes dedicated to be master nodes.

To demonstrate the latter advantage, a test of a distribution of 256,400 files in a real file system on a parallel and distributed partition with different number of storage units (SU) has been done.

The number of files per storage unit whose master node (SU_M) is stored in that storage unit has been measured, and then calculated the standard deviation of the average number of files per storage unit. Table A.1 shows the standard deviation of the number of storage units (I/O nodes) used.

The results shown in the table show that this scheme is simple, and allows a good distribution of master nodes (SU_M) and blocks among all storage units by balancing the use of such storage units and, therefore, the I/O load.

Number of Storage Units	Standard deviation
4	0.43
8	0.56
16	0.39
32	0.23
64	0.15
128	0.11

Table A.1: *Standard deviation in the distribution of master nodes*

A.4.6 Renaming of files

Because the master node (SU_M) is defined in terms of the file name, when a user renames a file, the master node (SU_M) of this file may also change. Algorithm A.1 illustrates the steps in *Expand* to rename a file.

```
1 rename(oldname, newname) {
2         SU_M = f_M(oldname, ||SU||)
3         SU'_M = f_M(newname, ||SU||)
4         { move S_M from SU_M to SU'_M }
5         ∀ SU_i ∈ F_P : { rename oldname to newname }
6 }
```

Algorithm A.1: *File renaming operation in Expand*

This process is shown in Figure A.10. The only operation necessary to preserve the coherence of the master node for all files in *Expand* is moving the metadata.

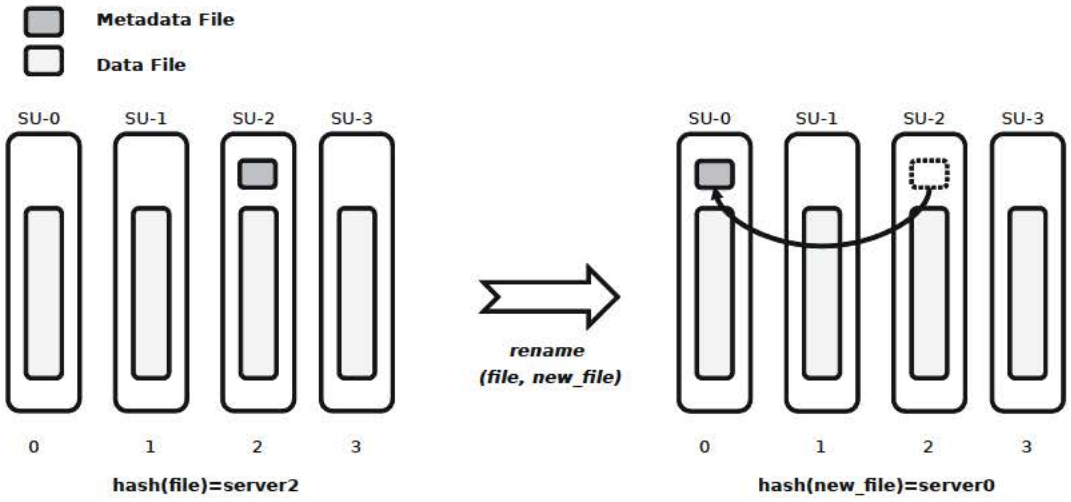


Figure A.10: *Process of renaming of files in Expand*

Initially the *base node*, which identifies the storage unit that stores the first data block of the file, is the same as the *master node*. But when the file name is changed, the *master node* can also change. That is when the *base node* and the *master node* may be different. This process does not affect the access to the data, since the metadata are still stored in the *base node*.

A.5 Directories

In addition to files, in a parallel file system there is another element of particular importance for users: directories. Thanks to directories users can group and organize their files.

In *Expand*, existing directories in the distributed partition (which is the logical view for the user of the data) are replicated to all storage units that form that partition.

As shown in Figure A.4, the existing directory tree in a distributed partition is replicated to all storage units.

When a directory is created in a distributed partition, a directory with the same name is created in all storage units. Removing a directory, or adding a new entry to a directory (either file or directory) is done in a similar way.

With a directory you can find all the entries located on it, and the attributes of those entries. Since directories are replicated in all storage units, you may distribute the information query load among the servers associated to storage units.

A.6 Virtual file handle and parallel access

Expand uses a virtual file handle to identify each parallel file internally. A file descriptor is an opaque reference to a file or directory that is independent of the file name.

The virtual file handle (VFH) is the reference used in *Expand* for the various operations performed on files or directories of the distributed partition. All operations in *Expand* use a virtual file handle. Such virtual file handle is defined as:

$$VFH = \bigcup_{i=1}^{\|SU\|} fh_i \in S_i,\ FH \in F_P \tag{A.24}$$

Where fh_i is the handle used by the i-th storage unit SU_i to access the data subfile S_i of the *Expand* file. Each SU_i used provides its own handler fh_i, which is dependent on the protocol and server used.

The file descriptor fh_i is a data structure that describes the file to the server, depending on the access protocol used. Therefore, for the NFS protocol, format and values associated with that descriptor may be different to that used in other protocols, such as FTP. Only the server can interpret the data contained within the file descriptor.

As mentioned, all operations in *Expand* use a virtual file descriptor. This virtual descriptor is the reference to the file used in all operations. When *Expand* needs to access to a subfile, it uses the appropriate file descriptor associated with the virtual descriptor.

To open a file, the fh_i of the metadata subfile S_M that resides in the storage unit SU_M is obtained, and it includes the metadata of the file. This metadata contain the servers and the blocks distribution policy for the file. Handlers of data subfiles are obtained on demand when access to data of a subfile is required. This speeds up the process of opening a file.

When *Expand* needs to access data in a subfile, it uses the handler used by the storage unit where the subfile resides. To improve the I/O, *Expand* performs the operations in parallel. To do this, when a request involves k storage units, *Expand* performes k operations in parallel using *threads*. So, when using NFS servers, a parallel operation of k servers takes place, and is divided into k individual operations, each of which uses RPC and the NFS protocol to access the appropriate subfile. Another example of parallel operations can be seen in the creation of a file in *Expand*, which involves the creation of multiple data subfiles. This process is shown in Figure A.11, which performs a read operation involving multiple data blocks and several SU.

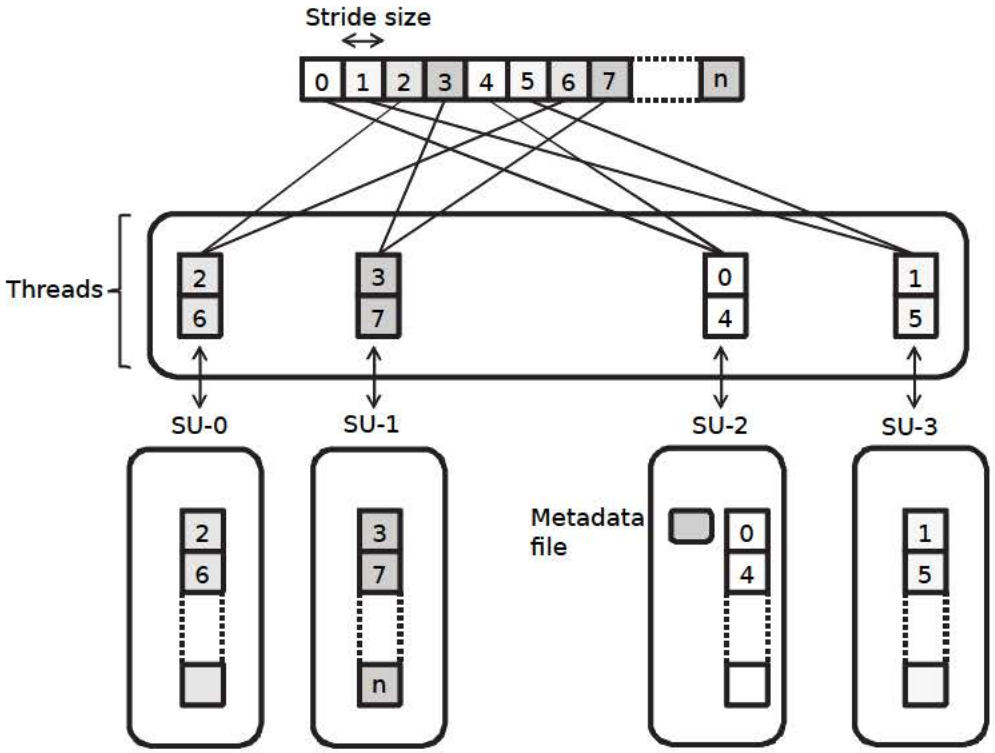


Figure A.11: *Parallel access to files in Expand*

A.7 Dynamic reconfiguration of partitions

Expand allows to reconfigure a partition dynamically by adding new server nodes to it. This way, the size of partitions can increase, and new resources can be added dynamically. The only problem introduced by the addition of a new node to an existing partition is the location of the subfile that stores the metadata of a file in the new *Expand* partition. Indeed, because the location of the *master* node of a file in the current implementation is based on the file name and the number of servers in the partition, when this number changes, so does the location of the metadata. The only operation required to keep the partition consistent is relocating the subfiles that stores metadata to their new *master* nodes. Figure A.12.a) shows the effect of adding a new node to a partition. The file in the old partition composed of three nodes had its metadata subfile in server 2. When adding a new node, the result of the metadata location function is server 3, which did not store the metadata previously. To keep this information consistent the subfile that stores the metadata moves from server 2 to server 3.

This operation can be performed in *Expand* in two ways: instantly to all files in the partition when one or more new server nodes are incorporated (as discussed in [Sánchez García (2009)], the cost required to perform this operation is not high), or delayed on demand when a file is opened. In this case the operation is performed at the opening of a file. If the search of the subfile that stores the metadata gives a wrong result, it performs the relocation of the subfile that stores the metadata to its new location transparently to the user. The new data to be added to an existing file will use the new incorporated nodes (see Figure A.12.b). For this type of reconfiguration, a new data distribution function is needed; a data distribution function that allows the use of different

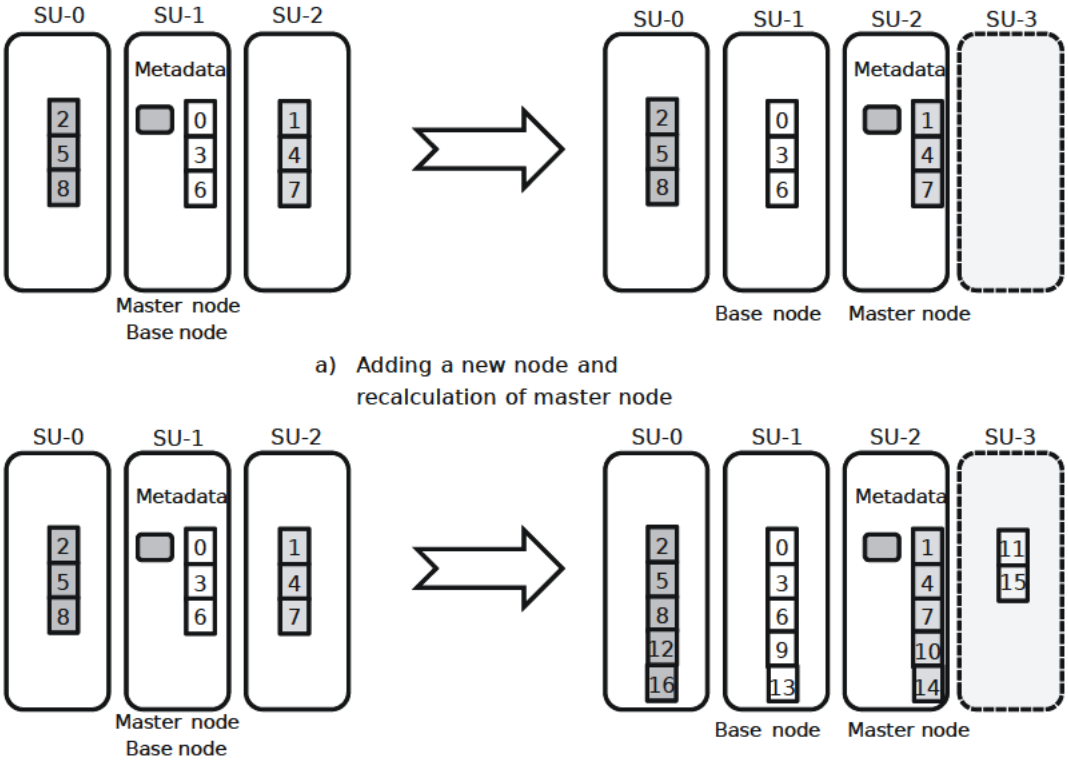


Figure A.12: *Adding a node to a partition in Expand*

data distribution patterns depending on whether the block is prior to the incorporation of data, or later. Figure A.12.b) shows that the blocks before the addition of the new I/O node use the previous data distribution pattern, while new data blocks use the new distribution pattern. These patterns are stored in the metadata file, which allows quick access to data, while it is necessary to control the modification of these patterns.

There is also the possibility of reconstructing a file or the entire partition when one or more nodes are incorporated. This process for a file is shown in Figure A.13. In this case, the cost of this process is higher as it involves the relocation of all data blocks of all files belonging to the partition. On the other hand, a unique data location function is provided for all data files in the partition, which prevents control over metadata.

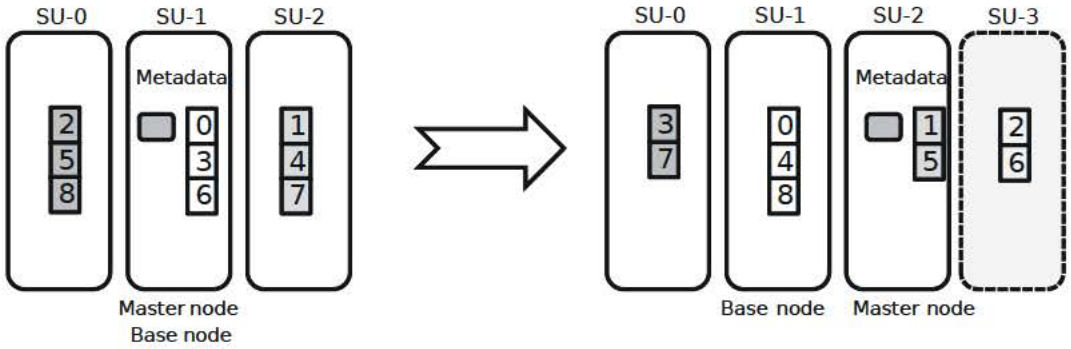


Figure A.13: *Reconstruction of a partition by adding a new node*

A.8 Access control and authentication

Access control and authentication depends on the individual protocol, but common to all is the need for:

- Identification: mapping of persons to users.
- Authentication: verification that the user is really who he/she represents himself/herself to be.
- Authorization: determination of what level of access a particular authenticated user should have. Control what operations can be performed with a given element, be it a file or directory.

For authentication and access control, *Expand* uses the services provided by the data server for access control and authentication of users on the system. Thus, for example, in the case of NFS, which is a stateless server and does not maintain information about files opened by users, the identity of the user is verified on each access to a file.

The Sun's RPC protocol requires clients to send their authentication information in each RPC request made to the server. This information is used by the server to check if the user can access the file. The current implementation for NFS in *Expand* uses the `AUTH_SYS` authentication method.

With this authentication method the server accesses the UID and GID of the user who is making the call to check if he/she have access to the file. The use of UID and GID implies that client and server must share the same user identification mapping. This requirement can be solved using a name service such as NIS, PAM, or LDAP.

For other distributed I/O protocols, such as FTP or GridFTP, authentication is required to start the communication. For example, in FTP it is necessary to use an identifier and an associated key, which must be previously registered in the server.

A.9 Architecture of *Expand*

Once the mechanisms for data access, and the management of data and metadata in *Expand* have been established, an overview of the design followed for building the architecture of Expand will be shown.

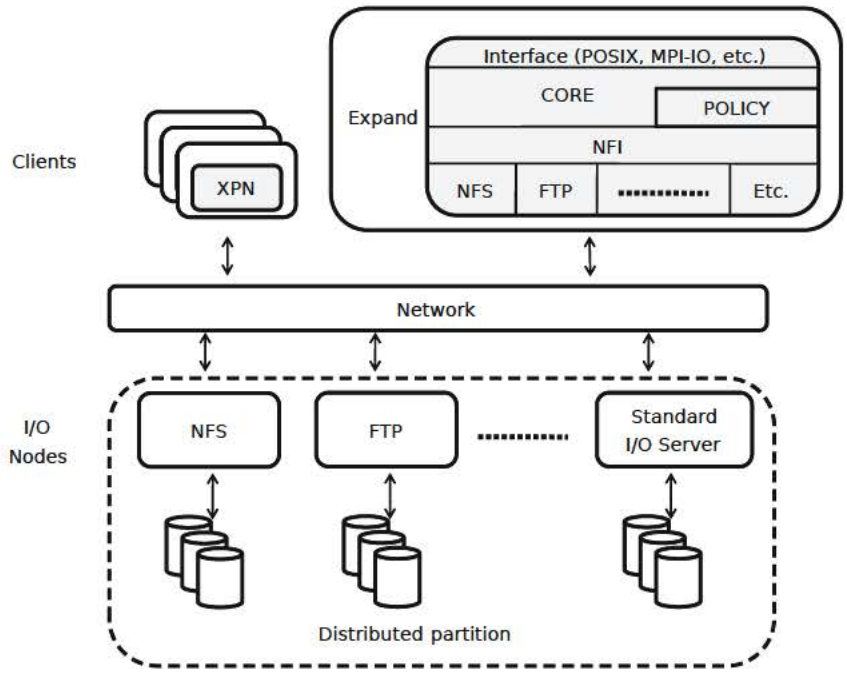


Figure A.14: *Expand Architecture*

The *Expand* architecture consists of three layers: *core*, *policy*, and *NFI* (see Figure A.14), each of which is designed to play a specific role for achieving the multiple tasks necessary to perform the I/O operations in *Expand*.

The main objectives of each of the layers are:

- *Core*: define the algorithms for managing the *Expand* parallel file system (data, metadata, directories, etc.), regardless of the protocol used in the data server.
- *Policy*: determine the policies to follow in various crucial aspects within *Expand*, such as the distribution of the data, metadata management, etc.
- *NFI*: abstracting communications with the I/O server, providing a single interface for data access.

Next, each of these layers of the *Expand* parallel file system will be detailed more in depth.

A.9.1 *Core* layer or file system kernel

This layer of the *Expand* architecture is responsible of:

- Performing the main operations of the system.
- Defining the basic algorithms for accessing and managing data.
- Exporting the basic user interfaces for using *Expand*.

This interface, similar to that available in POSIX, abstracts the developer from the complexity of the parallel file system.

Furthermore, this layer needs of the lower layers to perform the operations of data access, or to know the distribution of the data.

The operations performed at this level of abstraction are generic and independent of protocols and policies used at lower levels. These operations can be classified according to the scope of work:

- Operations for the management of the file system, such as system initialization and release of used resources (see Algorithms A.2, and A.3).

```
1 xpn_init() {
2         for each P_i ∈ P
3                 n = ||SU_Pi||
4                 P_i = {N_Pi, T_Pi, S_Pi, ||E_Pi||, ⋃_{j=1}^{n} {SU_j}}
5                 for each SU_ij ∈ P_i
6                         SU_ij = {C_Ei, I_Ei, D_Ei}
7 }
```

Algorithm A.2: *Initialization operation in Expand*

```
1 xpn_destroy() {
2         for each F_P
3                 {free resources of F_Pi}
4         for each P_i ∈ P
5                 {free resources of P_i}
```

Algorithm A.3: *Resources release operation in Expand*

- Operations that manage objects of the parallel file system (files and directories), allowing their creation (see Algorithm A.4), opening (see Algorithm A.5), and elimination from the system (see Algorithm A.6).
- Operations for data reading (see Algorithm A.7), or writing (see Algorithm A.8).

A.9.2 *Policy* layer or policy management

This layer is located in the middle level of the system, and is responsible for providing to the *core* layer the policies to be used in cases where the basic algorithms defined above depend on external data, or on policies defined by the user.

The operations performed are:

- Define the data distribution in I/O nodes (for example, *round-robin*).

```
1  xpn_creat(filename, mode) {
2          {obtain P of filename}
3          f_M(filename) → SU_M ∈ P
4          {create S_M in P}
5          for each SU ∈ P
6                  {create S_i}
7          for each S_i ∈ F_P
8                  {obtain fh_i ∈ S_i}
9          fd ← FH ∪ fh_i
10 }
```

Algorithm A.4: *File creation operation in Expand*

```
1  xpn_open(filename, flags [, mode]) {
2          {obtain P of filename}
3          f_M(filename) → SU_M ∈ P
4          for each SU ∈ P
5                  {obtain fh_i ∈ S_i}
6          {obtain S_M ∈ SU_M}
7          fd ← FH
8  }
```

Algorithm A.5: *File opening operation in Expand*

- Locate I/O nodes.
- Define the available nodes to be used (by default, all available nodes).
- Perform naming operations.
- Locate the metadata location function, etc.

This layer is also responsible for initializing the lower layers according to the protocol established in the configuration parameters (url, protocol, directories, etc.). To perform these tasks, this layer provides a generic interface that uses the kernel of the *Expand* file system.

A.9.3 *Network File Interface* layer or access to I/O servers

This layer, responsible for accessing data from different I/O servers, is called *Network File Interface*, or *NFI* for short. Mainly, it provides two critical elements in the architecture used by the upper layers:

- Sets a single interface for accessing data from the upper layers of the architecture, abstracting internal operations necessary to perform the low-level I/O operations, and management of the I/O servers.

```
1  xpn_unlink(filename) {
2          {obtain P of filename}
3          f_M(filename) → SU_M ∈ P
4          for each SU ∈ P
5                  {remove S_i ∈ F_P}
6          {remove S_M ∈ SU_M}
7  }
```

Algorithm A.6: *File removal operation in Expand*

```
1  xpn_read(fd, buffer, size) {
2          {obtain P of fd}
3          for each SU ∈ P
4                  if fh_i ∩ SU_i = ∅
5                          {obtain fh_i}
6          {divide buffer in {d, v} whose size ≤ ||B_ij|| ∈ P }
7          ∀{d, v} ∈ buffer
8                  f_d(d_k, v_k) ⇐ B_ij ∈ S_i
9  }
```

Algorithm A.7: *Parallel read operation in Expand*

- Defines the necessary mechanisms to allow parallel operations if parallelism has been set at higher levels of the architecture.

The interface provides various types of operations, both for handling elements (files or directories) of the file system of the I/O server, as for accessing data stored therein. On the other hand, management functions for the I/O server have been included in order to perform actions such as starting or finishing communications with the I/O servers, which are necessary for the proper functioning of the system.

The mechanism used by NFI to handle transactions on an I/O server is similar to that used in Linux with its Virtual File System (VFS) [Card et al. (1994)]. Previously, a module that implements all the operations necessary for the management of the file system of an I/O server must be available. This module replaces calls to the NFI interface so that, when upper layers perform generic NFI operations, these calls are made in the NFI module corresponding to the desired implementation.

In *Expand*, an NFI module for each used I/O server is established, so that the operations between client and servers are independent from each other. All communications between NFI independent modules are done through higher levels, thereby reducing system complexity.

```
1  xpn_write(fd, buffer, size) {
2          {obtain P of fd}
3          for each SU ∈ P
4                  if fh_i ∩ SU_i = ∅
5                  {obtain fh_i}
6          {divide buffer in {d,v} whose size ≤ ||B_i|| ∈ P}
7          ∀{d,v} ∈ buffer
8                  B_ij ∈ S_i ⇐ f_d(d_k, v_k)
9  }
```

Algorithm A.8: *Parallel write operation in Expand*

A.10 User interfaces

Expand offers different I/O interfaces: its own built-in interface, POSIX, `jExpand`, and MPI-IO. Next sections will detail these interfaces.

A.10.1 Built-in and POSIX interfaces

The built-in interface is similar to POSIX, and provides direct access to data stored in *Expand* partitions.

Applications can access *Expand* partitions through the integration of the *Expand* library with the operating system by developing a FUSE module (*Filesystem in Userspace*) [Szeredi (2013)]. As shown in Figure A.15, the module is located in user space, thereby reducing the performance of applications.

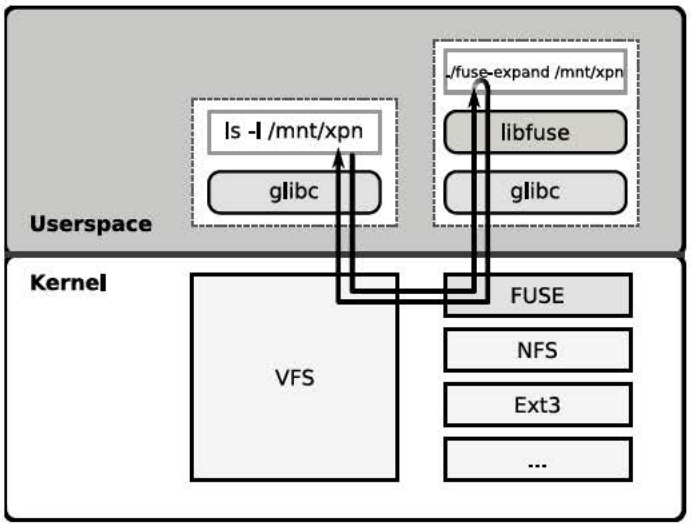


Figure A.15: *Expand integration in FUSE*

A.10.2 Java interface

An interface for Java applications is also provided. The Java interface is similar to that provided by the standard, using a class called `jExpand` [Pérez et al. (2005)] (see Figure A.16). This interface offers great advantages in heterogeneous environments since it can be used on any computer that has the Java virtual machine. Java applications that use *Expand* use the *Extended Filesystem API* developed by Sun [Sun Microsystems (2004)]. This API offers a common interface for multiple file system types. It is well suited for dynamically setting new file systems, and includes a set of classes similar to those found in `java.io` Java classes.

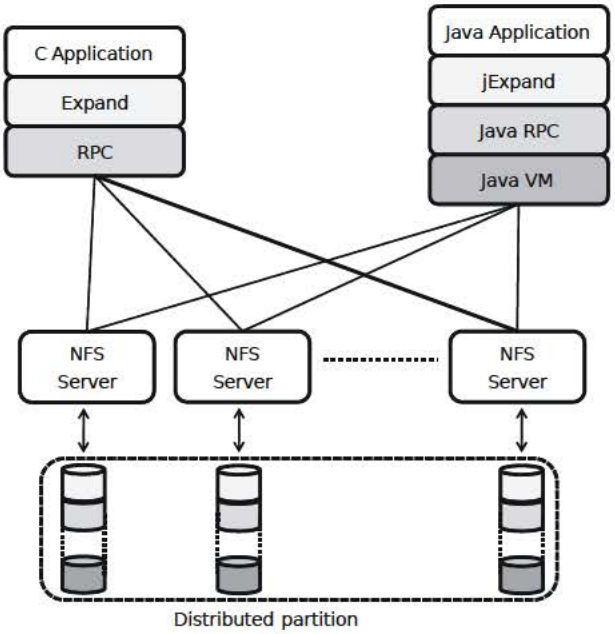


Figure A.16: *Expand implementation using C and Java for cluster environments*

A.10.3 MPI-IO interface

Although any standard C or Java application can benefit from *Expand* thanks to previous interfaces, they are not suitable for parallel applications using distribution patterns with small access sizes [Nieuwejaar and Kotz (1996a)].

Parallel applications can also use *Expand* with MPI-IO [Calderón et al. (2002a,b)]. *Expand* has been integrated in ROMIO [Thakur et al. (1999a,c)] and can be used with MPICH.

Portability in ROMIO is achieved using an abstract interface called ADIO (*Abstract-Device Interface for IO*). Therefore, to integrate *Expand* in ROMIO it has been necessary to implement an ADIO interface to perform the requests to the *Expand* parallel file system.

Parallel applications usually perform many small I/O requests (of the order of a kilobyte or less) [Thakur et al. (1999b)]. These small requests are the result of the combination of two factors:

- In many parallel applications, each process needs to access a large number of small pieces of data that are not located in the file contiguously.

- Many parallel file systems have a Unix interface (API), which allows a user to access only one contiguous piece of data of a file at any given time. Thus, non-contiguous access involves multiple calls for each piece of contiguous data.

With this interface, the file system can not easily detect the global access pattern of an individual process, or group of processes working together. As a result, the file system is constrained in the optimizations it can do.

Many file systems offer their own extensions or variations of Unix interface, but these variations make programs not portable.

To overcome these limitations in performance and portability of parallel I/O interfaces, the *MPI-Forum* defined a new interface for parallel I/O as part of the MPI-2 standard. This interface is known as *MPI-IO*.

There are multiple implementations of MPI-IO, which facilitates applications portability. Furthermore, MPI-IO contains a functionality-rich interface, which allows users to indicate non-contiguous access patterns, so that reading or writing of data is performed in a single I/O request. It also allows to indicate collective I/O requests, to be performed by a group of processes.

MPI-IO also allows MPI applications to define *views* on a file. To do this, three elements are defined:

- An *etype*, which is the access and positioning unit.
- A *filetype*, which is the basis for the division of a file among processes.
- A template to access the file.

A *filetype* can be formed by a simple data type, or a derived MPI datatype constructed from multiple instances of the same *etype*. A *view* establishes the actual, visible, and accessible data set of a file as an ordered set of *etypes*. Each process has its own *view*, consisting of three values: an offset, an *etype*, and a *filetype*. The pattern described by the *filetype* is repeated, starting at the indicated offset, to define the *view*.

This interface has been included in *Expand* [Calderón et al. (2002a,b)] through its integration in ROMIO [Thakur et al. (1999a)], and can be used with the MPICH distribution (see Figure A.17). Portability is achieved in ROMIO by an abstract interface of parallel I/O called ADIO [Corbett et al. (1995)].

ROMIO [Thakur et al. (1999a,c)] is an implementation of MPI-IO, which uses an abstract interface of I/O devices called ADIO (*Abstract Device Input Output*). ROMIO uses ADIO to achieve the desired portability.

ADIO [Thakur et al. (1999a,c)] is a mechanism specifically designed to implement a portable parallel I/O API on multiple file systems. ADIO consists of a small set of basic functions for parallel I/O.

ROMIO implements the following optimizations:

- *Data sieving*, (see Figure A.18) that transforms a sequence of small readings which are close to each other in a single read operation that covers all. Once all data are available in memory, those needed are taken. This sieving process allows to optimize the access to the parallel storage system, which is generally optimized for large requests.

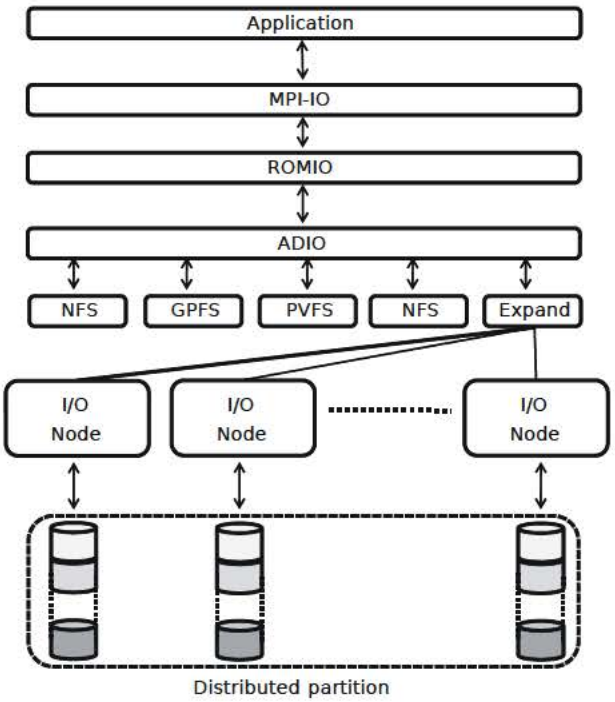


Figure A.17: *Expand integration in ROMIO*

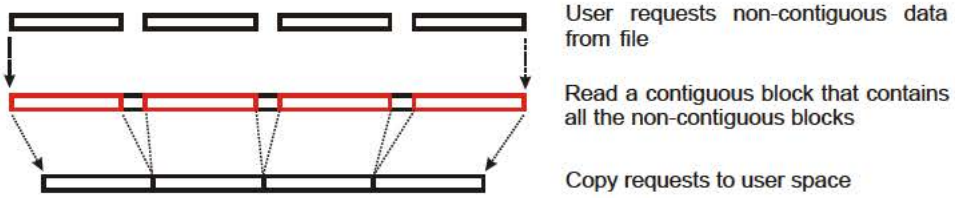


Figure A.18: *Data sieving*

This technique should be used properly to not consume too much memory.

- *Collective I/O*, which allows to mix requests of different processes, when they overlap in areas of the file that they are accessing. In this way, a request to the storage system can serve several requests from different processes.

- *Generalized Two-Phase I/O*, which allows a process of a group of processes to perform the I/O operations for the whole set of processes. In a first phase all requests are grouped. In the second phase this process sends the data for each of the members of the group, being an intermediary.

Figure A.19 shows the integration of *Expand* within ROMIO. This integration of *Expand* required several phases. The main phases are:

- Implement the ADIO interface for *Expand*.

- Connect the ADIO for *Expand* to ROMIO.

- Modify the MPICH compilation process to include the ADIO of *Expand* in such compilation process.

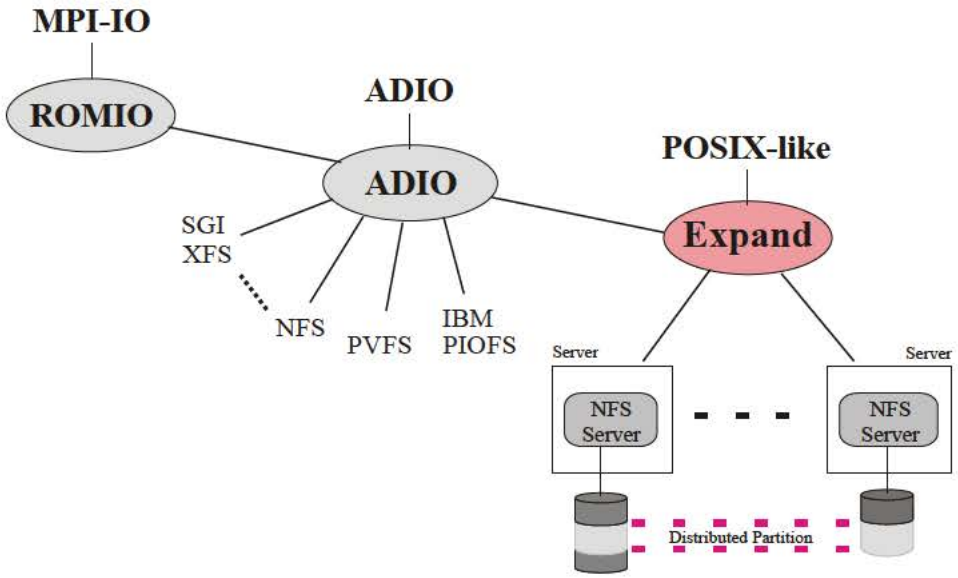


Figure A.19: *Expand integration in ROMIO*

Implementing the ADIO interface for *Expand* includes the implementation of various basic functions, as well as the possibility of implementing a set of optional features. These optional parts allows a greater degree of freedom in providing some optimized versions. Otherwise, ADIO implements default versions.

Specifically, the necessary functionality comprises:

- *Opening a file.* All file open operations are considered collective operations.
- *Closing a file.* The closing operation is also a collective operation.
- *Contiguous reads and writes.* ADIO provides separate routines for contiguous and non-contiguous accesses.
- *Non-contiguous reads and writes.* Parallel applications often need to read or write data which are located non-contiguously on files, and even on memory. ADIO provides routines to specify non-contiguous accesses with a single invocation.
- *Non-blocking reads and writes.* ADIO provides non-blocking versions of all read and write calls.
- *Collective reads and writes.* A collective routine must be called by all processes belonging to a group that opened the file.
- *Seek.* This function can be used to change the position of an individual file pointer.
- *Test and wait.* These operations are used to test the completion of non-blocking operations.

- *File Control*. This operation is used to set or get information about an opened file.
- *Miscellaneous*. Other operations included in ADIO provide routines to delete files, resize them, write cached data to disk, and initialize and finalize ADIO.

The result of the integration is a new version of MPICH which includes support for *Expand*, so that any parallel application can use *Expand* services through the MPI-IO interface.

For example, in the case of the example included in the ROMIO distribution named *simple*, to run the application using *Expand*, the following should be indicated:

```
$ mpirun -np 4 simple -fname xpn:/xpn1/example-02
```

That is, it is only necessary to include the prefix `xpn:`, that identifies a file of an *Expand partition*, namely the partition identified as `xpn1`.

A.11 Summary

In this chapter we have presented the architecture, design, and implementation of the *Expand* parallel file system, which is taken as a basis in this work for a parallel file system for large-scale distributed systems.

The architecture of the *Expand* parallel file system, the distributed partition model, and the parallel file model have been defined. Also, the name system, as well as the data access, or the metadata management have been described. Moreover, access control and authentication methods, used for access to data arranged in parallel partitions, have been detailed. Finally, interfaces used by the architecture (POSIX, MPI-IO, etc.) were detailed.

The main motivation is to provide a parallel file system for heterogeneous *clusters* using standard servers that do not require modification. *Expand* is built using standard data servers as a basis. This solution is very flexible since it is not necessary to install new servers to run *Expand*. It is also independent of the operating system used on clients because it uses the protocol that provides the data server itself.

The reasons for using *Expand* as a model in this work is that it is generic enough, and it offers a similar architecture to other parallel file systems for *clusters*. In addition, the fact that it has no centralized metadata manager, and that has the directory tree replicated allows a better adaptation to different types of computing environments.

Bibliography

L. A. Adamic and B. A. Huberman. The web's hidden order. *Comm. ACM*, 44(9):55–59, Sep 2001.

S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986. ISSN 0018-9162.

B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, and I. Foster. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *Mass Storage Systems and Technologies, 2001. MSS '01. Eighteenth IEEE Symposium on Mass Storage Systems and Technologies*, pages 13–13, 2001a. DOI: 10.1109/MSS.2001.10001. URL http://toolkit.globus.org/alliance/publications/papers/msc01.pdf.

B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, May 2002. ISSN 0167-8191. DOI: 10.1016/S0167-8191(02)00094-7. URL http://toolkit.globus.org/alliance/publications/papers/dataMgmt.pdf.

W. Allcock, A. Chervenak, I. Foster, L. Pearlman, V. Welch, and M. Wilde. Globus toolkit support for distributed data-intensive science. In *International Conference on Computing in High Energy and Nuclear Physics, Beijing, China*, 2001b. URL http://toolkit.globus.org/alliance/publications/papers/Globus.CHEP01.pdf.

W. Allcock, J. Bester, J. Bresnahan, S. Meder, and S. Tuecke. GridFTP: Protocol Extensions to FTP for the Grid, 2003. URL http://www.ggf.org/documents/GFD.20.pdf.

W. Allcock, J. Bresnahan, R. Kettimuthu, and J. Link. The Globus eXtensible Input/Output System (XIO): A protocol independent IO system for the Grid. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8 pp.–, April 2005. DOI: 10.1109/IPDPS.2005.429.

American Physical Society. Einstein@home. Last visited, Apr. 2013. URL http://www.einsteinathome.org.

T. Amjad, M. Sher, and A. Daud. A survey of dynamic replication strategies for improving data availability in data grids. *Future Generation Computer Systems*, 28(2):337 – 349, 2012. ISSN 0167-739X. DOI: 10.1016/j.future.2011.06.009. URL http://www.sciencedirect.com/science/article/pii/S0167739X11001208.

D. Anderson, E. Korpela, and R. Walton. High-performance task distribution for volunteer computing. In *e-Science and Grid Computing, 2005. First International Conference on*, pages 8 pp.–203, July 2005. DOI: 10.1109/E-SCIENCE.2005.51.

D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2256-4. DOI: 10.1109/GRID.2004.14.

D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002. ISSN 0001-0782. DOI: 10.1145/581571.581573.

T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM SIGOPS Operating Systems Review*, 29(5):109–126, 1995.

T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems (TOCS) - Special issue on operating system principles*, 14(1):41–79, Feb. 1996. ISSN 0734-2071. DOI: 10.1145/225535.225537. URL http://doi.acm.org/10.1145/225535.225537.

G. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 1999.

S. Androutsellis-Theotokis and D. Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Comput. Surv.*, 36(4):335–371, Dec. 2004. ISSN 0360-0300. DOI: 10.1145/1041680.1041681. URL http://doi.acm.org/10.1145/1041680.1041681.

Argonne National Laboratory. The parallel virtual file system 2 (PVFS2), Apr. 2011. URL http://www.pvfs.org/.

M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pages 126–137, May 1996.

L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010. ISSN 1389-1286. DOI: 10.1016/j.comnet.2010.05.010. URL http://www.sciencedirect.com/science/article/pii/S1389128610001568.

Avaki Corporation. Avaki home page, 2005. URL http://www.avaki.com.

R. Bagrodia, R. Meyer, M. Takai, Y.-A. Chen, X. Zeng, J. Martin, and H. Y. Song. Parsec: a parallel simulation environment for complex systems. *Computer*, 31(10):77–85, Oct. 1998. ISSN 0018-9162. DOI: 10.1109/2.722293.

S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zapalla. Improving simulation for network research. Technical Report 99-702, University of Southern California, Computer Science Department, Mar. 1999.

P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pages 151–160, Jun 1998.

C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of CASCON*, 1998.

A. Bassi, M. Beck, G. Fagg, T. Moore, J. S. Plank, and et al. The Internet Backplane Protocol: A Study in Resource Sharing. In *Future Generation Computing Systems*, pages 551–561, 2002.

A. Bassi, H. Europe, and G. Horn. Internet of things in 2020, roadmap for the future. Technical report, INFSO D.4 Networked Enterprise & RFID INFSO G.2 Micro & Nanosystems, in co-operation with the RFID Working Group of the European Technology Platform on Smart Systems Integration (EPoSS), Sept. 2008. URL http://www.smart-systems-integration.org/public/documents/publications/Internet-of-Things_in_2020_EC-EPoSS_Workshop_Report_2008_v3.pdf.

A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS 2009)*, volume 0, pages 1–8, Los Alamitos, CA, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5160922. URL http://doi.ieeecomputersociety.org/10.1109/IPDPS.2009.5160922.

A. Beck. High throughput computing: An interview with miron livny, 1997. URL http://www.hpcwire.com/hpc-bin/artread.pl?direction=Current&articlenumber=11444.

J. I. Beiriger, H. P. Bivens, S. L. Humphreys, W. R. Johnson, and R. E. Rhea. Constructing the asci computational grid. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, HPDC '00, pages 193–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0783-2. URL http://dl.acm.org/citation.cfm?id=822085.823324.

A. Beloglazov, S. F. Piraghaj, M. Alrokayan, and R. Buyya. Deploying OpenStack on CentOS using the KVM hypervisor and GlusterFS distributed file system. Technical Report CLOUDS-TR-2012-3, Cloud Computing and Distributed Systems Laboratory, The University of Melbourne, Aug. 2012.

B. Bergua, F. García Carballeira, L. M. Sánchez, A. Calderón, and J. Carretero. Adaptación del sistema de ficheros paralelo expand a entornos grid. In *XVIII Jornadas de Paralelismo*, Sept. 2007. ISBN 84-9732-593-6.

B. Bergua, F. García Carballeira, A. Calderón, L. M. Sánchez, and J. Carretero. Comparing Grid Data Transfer Technologies in the Expand Parallel File System. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, PDP '08, pages 110–114, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3089-5. DOI: 10.1109/PDP.2008.51. URL http://dx.doi.org/10.1109/PDP.2008.51.

B. Bergua, F. García Carballeira, A. Calderón, L. M. Sánchez, and J. Carretero. Improving the performance of the BOINC volunteer computing platform using the *Expand* parallel file system

(poster). In *Fifth IEEE International Conference on e-Science*. IEEE Computer Society, Dec. 2009a. Poster session.

B. Bergua, F. García Carballeira, L. M. Sánchez, A. Calderón, A. Rodríguez, and J. Carretero. Architecture for improving data transfers in grid using the expand parallel file system. In *3rd Iberian Grid Infrastructure Conference (IBERGRID 2009)*, pages 315–326, May 2009b. ISBN 978-84-9745-406-3.

B. Bergua, F. García Carballeira, A. Calderón, L. M. Sánchez, and J. Carretero. Mejora del entorno de computación voluntaria BOINC usando el sistema de ficheros paralelo Expand. In *XXI Jornadas de Paralelismo (JP2010)*, Sept. 2010.

B. Bergua Guerra. Adaptación del sistema de ficheros paralelo *Expand* 2.0 a entornos Grid. Bachelor's thesis, Computer Science department, Universidad Carlos III de Madrid, Sept. 2006.

F. Berman, G. C. Fox, and A. J. G. Hey, editors. *Grid Computing: Making the Global Infrastructure a Reality*, chapter 10, From Legion to Avaki: The Persistence of Vision, pages 265–298. John Wiley & Sons, March 2003. ISBN 0-470-85319-0.

T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

D. Bertsekas and R. Gallager. *Data networks (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 0-13-200916-1. URL http://web.mit.edu/dimitrib/www/datanets.html.

J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, IOPADS '99, pages 78–88, New York, NY, USA, 1999. ACM. ISBN 1-58113-123-2. DOI: 10.1145/301816.301839. URL http://doi.acm.org/10.1145/301816.301839.

L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *18th IEEE INFOCOM*, volume 1, pages 126–134, Mar 1999.

S. Brunett, K. Czajkowski, S. Fitzgerald, I. Foster, A. Johnson, C. Kesselman, J. Leigh, and S. Tuecke. Application experiences with the globus toolkit, 1998.

J. J. Bunn and H. B. Newman. *Grid Computing: Making the Global Infrastructure a Reality*, chapter 39: Data-intensive Grids for high-energy physics. John Wiley & Sons, Inc., 2003.

R. Buyya and S. Vazhkudai. Compute Power Market: Towards a Market-Oriented Grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, pages 574–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1010-8. URL http://dl.acm.org/citation.cfm?id=560889.792354.

A. Calderón. *Técnicas de tolerancia a fallos en sistemas de ficheros paralelos para clusters*. PhD thesis, Computer Science department, Universidad Carlos III de Madrid, 2005.

A. Calderón, F. García, J. Carretero, J. M. Pérez, and J. Fernández. An implementation of mpi-io on expand: A parallel file system based on nfs servers. In D. KranzlmÃijller, J. Volkert, P. Kacsuk, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message*

Passing Interface, volume 2474 of *Lecture Notes in Computer Science*, pages 306–313. Springer Berlin Heidelberg, 2002a. ISBN 978-3-540-44296-7. DOI: 10.1007/3-540-45825-5_47. URL http://dx.doi.org/10.1007/3-540-45825-5_47.

A. Calderón, F. García, J. Carretero, J. M. Pérez, and J. Fernández. An implementation of MPI-IO on Expand: A parallel file system based on NFS servers. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 306–313, London, UK, 2002b. Springer-Verlag. ISBN 3-540-44296-0.

A. Calderón, F. García Carballeira, J. Carretero, J. M. Pérez., and J. Fernández. Soporte de tolerancia a fallos en expand. In *XIV Jornadas de Paralelismo*, 2003.

D. G. Cameron, R. Carvajal-Schiaffino, A. P. Millar, C. Nicholson, K. Stockinger, and F. Zini. OptorSim: A simulation tool for scheduling and replica optimisation in data grids. In *Proceedings of the Computing in High Energy and Nuclear Physics (CHEP) conference*, 2004.

R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, 1994. ISBN 90-367-0385-9.

P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327. USENIX Association, 2000. ANL/MCS-P804-0400.

J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. ParFiSys: A parallel file system for MPP. *ACM Operating Systems Review, 30(2):74–80*, 1996.

J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. Performance increase mechanisms for parallel and distributed file systems. *Parallel Computing: Special Issue on Parallel I/O Systems. Elsevier*, 23(3):525–542, 1997.

H. Casanova and L. Marchal. A Network Model for Simulation of Grid Application. Rapport de recherche RR-4596, INRIA, Oct. 2002. URL http://hal.inria.fr/inria-00071989.

H. Casanova, A. Legrand, and M. Quinson. SimGrid: a generic framework for large-scale distributed experiments. In *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, UKSIM '08, pages 126–131, Washington, DC, USA, Mar. 2008. IEEE Computer Society. ISBN 978-0-7695-3114-4. DOI: 10.1109/UKSIM.2008.28. URL http://dx.doi.org/10.1109/UKSIM.2008.28.

H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014. URL http://hal.inria.fr/hal-01017319.

C. Catlett. In search of gigabit applications. *Communications Magazine, IEEE*, 30(4):42 –51, april 1992. ISSN 0163-6804. DOI: 10.1109/35.135788.

S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill Computer Science Series. McGraw-Hill, 1984. ISBN 9780070108295. URL http://books.google.es/books?id=sepQAAAAMAAJ.

CERN. Grid file access library 2.0. Last visited, Nov. 2015. URL https://dmc.web.cern.ch/projects/gfal-2/home.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267308.1267323.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008. ISSN 0734-2071. DOI: 10.1145/1365815.1365816. URL http://doi.acm.org/10.1145/1365815.1365816.

S. J. Chapin, D. Katramatos, J. Karpovich, and A. S. Grimshaw. The Legion Resource Management System. In *Proceedings of the 5 th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 162–178. Springer Verlag, 1999.

J. S. Chase, D. C. Anderson, and A. M. Vahdat. Interposed request routing for scalable network storage. In *Fourth Symposium on Operating System Design and Implementation (OSDI2000)*, pages 259–272, October 2000.

A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 23:187–200, 1999.

A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, and B. Tierney. Giggle: A framework for constructing scalable replica location services. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. URL http://dl.acm.org/citation.cfm?id=762761.762798.

D. M. Chiu. Some observations on fairness of bandwidth sharing. In *Proceedings of the Fifth IEEE Symposium on Computers and Communications (ISCC 2000)*, ISCC '00, pages 125–131, Washington, DC, USA, July 2000. IEEE Computer Society. ISBN 0-7695-0722-0. DOI: 10.1109/ISCC.2000.860626. URL http://dl.acm.org/citation.cfm?id=844383.845579.

S. Choi, R. Buyya, H. Kim, E. Byun, M. Baik, J. Gil, and C. Park. A taxonomy of desktop grids and its mapping to state-of-the-art systems. Technical Report GRIDS-TR-2008-3, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, Feb. 2008. URL http://www.cloudbus.org/reports/DesktopGridTaxonomy2008.pdf.

N. P. Chue Hong, M. Drescher, A. Krause, M. S. Memon, and M. Morgan. OGSA ByteIO implementations – experiences document. Technical Report GFD-E.146, Open Grid Forum, Mar. 2009. URL http://www.ogf.org/documents/GFD.146.pdf.

Cluster File Systems Inc. Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, November 2002. URL http://www.lustre.org/docs/whitepaper.pdf.

J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387905.

P. Corbett, S. Johnson, and D. Feitelson. Overview of the vesta prallel file system. *ACM Computer Architecture News*, 21(5):7–15, Dec. 1993.

P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, 1995.

F. Costa, L. Silva, I. Kelley, and G. Fedak. Optimizing the data distribution layer of boinc with bittorrent. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008. DOI: 10.1109/IPDPS.2008.4536446.

G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems (4th ed.): concepts and design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

J. Cowie, H. Liu, J. Liu, D. Nicol, and A. Ogielski. Towards realistic million-node internet simulations. In H. R. Arabnia, editor, *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 2129–2135. CSREA Press, June 1999a. ISBN 1-892512-15-7.

J. H. Cowie, D. M. Nicol, and A. T. Ogielski. Modeling the global internet. *Computing in Science & Engineering*, 1(1):42–50, Jan. 1999b. ISSN 1521-9615. DOI: 10.1109/5992.743621.

M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Trans. Networking*, 5(6):835–846, Dec 1997.

W. Dargie. *Fundamentals of Wireless Sensor Networks*. Wiley, New York, 2010. ISBN 9780470997659.

B. D. Davison. A Web Caching Primer. *IEEE Internet Computing*, 5(4):38–45, July 2001. ISSN 1089-7801. DOI: 10.1109/4236.939449. URL http://dx.doi.org/10.1109/4236.939449.

J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 137–150, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-0782. DOI: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

J. Dean and S. Ghemawat. MapReduce: A flexible data processing tool. *Communications of the ACM*, 53(1):72–77, Jan. 2010. ISSN 0001-0782. DOI: 10.1145/1629175.1629198. URL http://doi.acm.org/10.1145/1629175.1629198.

J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally Distributed Content Delivery. *IEEE Internet Computing*, 6(5):50–58, Sept. 2002. ISSN 1089-7801. DOI: 10.1109/MIC.2002.1036038. URL http://dx.doi.org/10.1109/MIC.2002.1036038.

distributed.net. distributed.net. Last visited, Apr. 2013. URL http://distributed.net.

S. Dolev. *Self-Stabilization*. MIT Press, 2000. ISBN 0-262-04178-2.

B. Donassolo, H. Casanova, A. Legrand, and P. Velho. Fast and scalable simulation of volunteer computing systems using SimGrid. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 605–612, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. DOI: 10.1145/1851476.1851565. URL http://doi.acm.org/10.1145/1851476.1851565.

G. Donvito, G. Marzulli, and D. Diacono. Testing of several distributed file-systems (hdfs, ceph and glusterfs) for supporting the hep experiments analysis. *Journal of Physics: Conference Series*, 513(4):042014, 2014. URL http://stacks.iop.org/1742-6596/513/i=4/a=042014.

A. B. Downey. The structural cause of file size distributions. In *9th Modeling, Anal. & Simulation of Comput. & Telecomm. Syst. (MASCOTS)*, Aug 2001.

H. Duan, S. Yu, M. Mei, W. Zhan, and L. Li. Cstore: A desktop-oriented distributed public cloud storage system. *Computers & Electrical Engineering*, 42:60 – 73, 2015. ISSN 0045-7906. DOI: 10.1016/j.compeleceng.2014.11.001. URL http://www.sciencedirect.com/science/article/pii/S0045790614002705.

C. L. Dumitrescu and I. Foster. GangSim: A simulator for grid scheduling studies. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, CCGRID '05, pages 1151–1158, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-9074-1. URL http://dl.acm.org/citation.cfm?id=1169223.1169632.

H. Eckardt. Investigation of distributed disk I/O concepts. Technical report, ESPRIT PUMA, Siemens, 1990.

EGEE Project. gLite. Last visited, Apr. 2013. URL http://glite.cern.ch/.

J. Ekanayake, T. Gunarathne, and J. Qiu. Cloud technologies for bioinformatics applications, 2010.

European Middleware Initiative. Gfal functional description. Last visited, Apr. 2013a. URL https://svnweb.cern.ch/trac/lcgutil/wiki/GFAL.

European Middleware Initiative. Grid file access library 2.0. Last visited, Apr. 2013b. URL https://svnweb.cern.ch/trac/lcgutil/wiki/gfal2.

European Middleware Initiative. Grid file access library 2.0. Last visited, Nov. 2015. URL http://www.eu-emi.eu/emi-2-matterhorn-products/-/asset_publisher/B4Rk/content/gfal-lcg-util.

D. G. Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, May 2015. URL http://www.cs.huji.ac.il/~feit/wlmod/.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999.

R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. URL http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, May 2002. ISSN 1533-5399. DOI: 10.1145/514183.514185. URL http://doi.acm.org/10.1145/514183.514185.

A. Finkelstein, C. Gryce, and J. Lewis-Bowen. Relating Requirements and Architectures: A Study of Data-Grids. *Journal of Grid Computing*, 2:207–222, 2004. ISSN 1570-7873. DOI: 10.1007/s10723-004-6745-6. URL http://dx.doi.org/10.1007/s10723-004-6745-6.

S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transactions on Networking (TON)*, 7(4):458–472, Aug. 1999. ISSN 1063-6692. DOI: 10.1109/90.793002. URL http://dx.doi.org/10.1109/90.793002.

I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *Network and Parallel Computing, LNCS*, 3779/2005:2–13, 2005a. ISSN 0302-9743 (Print) 1611-3349 (Online). DOI: 10.1007/11577188_2. URL http://dx.doi.org/10.1007/11577188_2.

I. Foster. A globus toolkit primer, 2005b. URL http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf.

I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

I. Foster and C. Kesselman. *The grid: blueprint for a new computing infrastructure*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Elsevier, 2004. ISBN 9781558609334. URL http://books.google.es/books?id=8-0BofIhoU0C.

I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15:200–222, August 2001. ISSN 1094-3420. DOI: 10.1177/109434200101500302. URL http://dl.acm.org/citation.cfm?id=1080644.1080667.

I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002a. URL http://www.globus.org/alliance/publications/papers/ogsa.pdf.

I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35:37–46, June 2002b. ISSN 0018-9162. DOI: 10.1109/MC.2002.1009167. URL http://dx.doi.org/10.1109/MC.2002.1009167.

I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. *Grid Computing: Making the Global Infrastructure a Reality*, chapter The Physiology of the Grid, pages 217–249. Wiley, 2003.

K. Fujiwara. Cost and accuracy of packet-level vs. analytical network simulations: an empirical study. Master's thesis, University of Hawai'i, May 2007.

K. Fujiwara and H. Casanova. Speed and accuracy of network simulation in the SimGrid framework. In *Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, ValueTools '07, pages 12:1–12:10, ICST, Brussels, Belgium, Belgium, Oct. 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-00-4. URL http://dl.acm.org/citation.cfm?id=1345263.1345279.

F. García Carballeira, A. Calderón, J. Carretero, J. Fernández, and J. M. Pérez. Parallel file system based on NFS servers for heterogeneous clusters. In *3rd ACIS International Conference on Software Engineering, Artificial Intelligence Networking and Parallel/Distributed Computing, SNPD02*, 2002.

F. García Carballeira, A. Calderón, J. Carretero, J. Fernández, and J. M. Pérez. The design of the Expand parallel file system. *International Journal of High Performance Computing Applications*, 17(1):21–37, 2003a. DOI: 10.1177/1094342003017001003. URL http://hpc.sagepub.com/content/17/1/21.abstract.

F. García Carballeira, A. Calderón, J. Carretero, J. M. Pérez, and J. Fernández. An expandable parallel file system using NFS servers. In *Proceedings of the 5th international conference on High performance computing for computational science*, VECPAR'02, pages 565–578, Berlin, Heidelberg, 2003b. Springer-Verlag. ISBN 3-540-00852-7. URL http://dl.acm.org/citation.cfm?id=1766851.1766897.

F. García Carballeira, A. Calderón, J. Carretero, J. M. Pérez, and J. Fernández. A parallel and fault tolerant file system based on NFS server. In *Euromicro Conference on Parallel Distributed and Network based Processing*, 2003c.

F. García Carballeira, J. Carretero, A. Calderón, J. D. García, and L. M. Sánchez. A global and parallel file system for grids. *Future Generation Computer Systems*, 23(1):116–122, Jan. 2007. ISSN 0167-739X. DOI: 10.1016/j.future.2006.06.004. URL http://www.sciencedirect.com/science/article/pii/S0167739X06001282.

S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003a. ACM. ISBN 1-58113-757-5. DOI: 10.1145/945445.945450. URL http://doi.acm.org/10.1145/945445.945450.

S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Operating Systems Review*, 37:29–43, Oct. 2003b. ISSN 0163-5980. DOI: 10.1145/1165389.945450. URL http://doi.acm.org/10.1145/1165389.945450.

S. Ghosh. *Distributed Systems: An Algorithmic Approach*. Chapman & Hall/CRC, 2006.

G. A. Gibson. The Scotch paralell storage systems. Technical Report CMU-CS-95-107, Scholl of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1995.

D. Giusto, A. Iera, G. Morabito, and L. Atzori. *The Internet of Things: 20th Tyrrhenian workshop on digital communications*. Springer Science & Business Media, 2010. ISBN 978-1-4419-1673-0. DOI: 10.1007/978-1-4419-1674-7.

T. R. Henderson, S. Roy, S. Floyd, and G. F. Riley. ns-3 project goals. In *Proceeding from the 2006 workshop on ns-2: the IP network simulator*, WNS2 '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-508-8. DOI: 10.1145/1190455.1190468. URL http://doi.acm.org/10.1145/1190455.1190468.

T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. B. Kopena. Network simulations with the ns-3 simulator. In *Demonstrations of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM'08, Seattle, Washington, USA, Aug. 2008. ACM. ISBN 978-1-60558-175-0.

T. Hey and A. E. Trefethen. The UK e-Science Core Programme and the Grid. *Future Gener. Comput. Syst.*, 18(8):1017–1031, Oct. 2002. ISSN 0167-739X. DOI: 10.1016/S0167-739X(02)00082-1. URL http://dx.doi.org/10.1016/S0167-739X(02)00082-1.

M. Hill. *Readings in Computer Architecture*. Morgan Kaufmann, San Diego, 2000. ISBN 1558605398.

P. Horn. The IBM vision for autonomic computing. Technical report, IBM, 2001. URL http://www.research.ibm.com/autonomic/manifesto.

W. Hoschek, F. J. Jaén-Martínez, A. Samar, H. Stockinger, and K. Stockinger. Data Management in an International Data Grid Project. In *Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, GRID '00, pages 77–90, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-41403-7. URL http://dl.acm.org/citation.cfm?id=645440.652836.

J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, Feb. 1988. ISSN 0734-2071. DOI: 10.1145/35037.35059. URL http://doi.acm.org/10.1145/35037.35059.

D. Howe, M. Costanzo, P. Fey, T. Gojobori, L. Hannick, W. Hide, D. P. Hill, R. Kania, M. Schaeffer, S. St Pierre, S. Twigger, O. White, and S. Y. Rhee. The future of biocuration. *Nature*, 455(7209):47–50, Sep 04 2008. DOI: 10.1038/455047a.

K. Hwang, H. Jin, E. Chow, C.-L. Wang, and Z. Xu. Designing SSI clusters with hierarchical checkpointing and single I/O space. *IEEE Concurrency*, 7(1):60–69, 1999. ISSN 1092-3063. DOI: 10.1109/4434.749136.

Indiana University. Futuregrid portal. Last visited, Apr. 2013. URL https://portal.futuregrid.org/.

INRIA/IN2P3. XtremWeb: the open source platform for desktop grids, 2008. URL http://www.xtremweb.net/.

G. Irlam. Unix file size survey, 1993. URL http://www.gordoni.com/ufs93.html.

Jabber.org. Jabber. Last visited, Apr. 2013. URL http://www.jabber.org.

W. E. Johnston, D. Gannon, and B. Nitzberg. Grids as production computing environments: The engineering aspects of nasa's information power grid. In *Proceedings of the 8th IEEE*

International Symposium on High Performance Distributed Computing, HPDC '99, pages 34– , Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0287-3. URL http://dl.acm.org/citation.cfm?id=822084.823281.

D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

F. P. Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8:33–37, 1997. URL http://www.statslab.cam.ac.uk/~{}frank/elastic.html.

C. Kesselman. Internet x.509 public key infrastructure proxy certificate profile. *IETF*, 2001.

R. Kettimuthu, L. Wantao, J. M. Link, and J. Bresnahan. A GridFTP Transport Driver for Globus XIO. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2008, Las Vegas, Nevada, USA, July 14-17, 2008, 2 Volumes*, pages 843–849, 2008. URL http://toolkit.globus.org/alliance/publications/papers/gridftp_transport_driver_xio.pdf.

S. U. Khan and I. Ahmad. Comparison and analysis of ten static heuristics-based internet data replication techniques. *Journal of Parallel and Distributed Computing*, 68(2):113 – 136, 2008. ISSN 0743-7315. DOI: 10.1016/j.jpdc.2007.06.009. URL http://www.sciencedirect.com/science/article/pii/S0743731507001153.

G. H. Kim, R. G. Minnich, and L. McVoy. Bigfoot-NFS: A Parallel File-Striping NFS Server, 1994.

J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, Feb. 1992. ISSN 0734-2071. DOI: 10.1145/146941.146942. URL http://doi.acm.org/10.1145/146941.146942.

K. Krauter, R. Buyya, and M. Maheswaran. A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing. *Softw. Pract. Exper.*, 32(2):135–164, Feb. 2002. ISSN 0038-0644. DOI: 10.1002/spe.432. URL http://dx.doi.org/10.1002/spe.432.

O. Krieger. *HFS: A Flexible File System for Shared-Memory Multiprocessors*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 1994.

B. Krishnamurthy, C. Wills, and Y. Zhang. On the Use and Performance of Content Distribution Networks. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, IMW '01, pages 169–182, New York, NY, USA, 2001. ACM. ISBN 1-58113-435-5. DOI: 10.1145/505202.505224. URL http://doi.acm.org/10.1145/505202.505224.

J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGARCH Comput. Archit. News*, 28(5):190–201, 2000. ISSN 0163-5964. DOI: 10.1145/378995.379239.

D. Kusnetzky. What is "Big Data?". ZDNet, Feb. 2010. URL http://www.zdnet.com/blog/virtualization/what-is-big-data/1708.

H. Lamehamedi, Z. Shentu, B. Szymanski, and E. Deelman. Simulation of dynamic data replication strategies in data grids. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 10–, 2003. DOI: 10.1109/IPDPS.2003.1213206.

S. M. Larson, C. D. Snow, M. R. Shirts, and V. S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. In R. P. Grant, editor, *Computational Genomics: Theory and Application*. Horizon Bioscience, 2004. ISBN 978-1-904933-01-4. URL http://books.google.com.do/books?id=f0JkQgAACAAJ.

B.-D. Lee and J. Weissman. Dynamic replica management in the service grid. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, pages 433–434, 2001. DOI: 10.1109/HPDC.2001.945213.

A. Legrand and J. Lerouge. MetaSimGrid: Towards realistic scheduling simulation of distributed applications. Technical Report 2002-28, Laboratoire de l'Informatique du Parallélisme (LIP), École Normale Supérieure de Lyon, July 2002.

J. Li, W. K. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 39, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 1-58113-695-1.

J. Li, M. Humphrey, D. A. Agarwal, K. R. Jackson, C. van Ingen, and Y. Ryu. escience in the cloud: A modis satellite data reprojection and reduction pipeline in the windows azure platform. In *IPDPS*, pages 1–10. IEEE, 2010. DOI: 10.1109/IPDPS.2010.5470418. URL http://dblp.uni-trier.de/db/conf/ipps/ipdps2010.html#LiHAJIR10.

LIGO Scientific Collaboration. Einstein@Home search for periodic gravitational waves in LIGO S4 data. *Physical Review D*, 79:022001, 2009. URL doi:10.1103/PhysRevD.79.022001.

LIGO Scientific Collaboration and D. P. Anderson. Einstein@Home search for periodic gravitational waves in early S5 LIGO data. *Physical Review D*, 80:042003, 2009. URL doi:10.1103/PhysRevD.80.042003.

W. Ligon and R. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the Extreme Linux Workshop*, June 1999.

B. Liu, D. Figueiredo, Y. Guo, J. Kurose, and D. Towsley. A study of networks simulation efficiency: fluid simulation vs. packet-level simulation. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1244–1253 vol.3, 2001. DOI: 10.1109/INFCOM.2001.916619.

X. Liu and A. A. Chien. Traffic-based load balance for scalable network emulation. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 40–, New York, NY, USA, 2003. ACM. ISBN 1-58113-695-1. DOI: 10.1145/1048935.1050190. URL http://doi.acm.org/10.1145/1048935.1050190.

T. Loukopoulos and I. Ahmad. Static and adaptive data replication algorithms for fast information access in large distributed systems. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 385–392, 2000. DOI: 10.1109/ICDCS.2000.840950.

S. H. Low. A duality model of TCP and queue management algorithms. *IEEE/ACM Transactions on Networking*, 11(4):525–536, Aug. 2003. ISSN 1063-6692. DOI: 10.1109/TNET.2003.815297. URL http://dx.doi.org/10.1109/TNET.2003.815297.

C. Lynch. How do your data grow? *Nature*, 455(7209):28–9, Sep 04 2008. DOI: 10.1038/455028a.

N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 1558603484.

T. M. Madhyastha. *Automatic classification of input/output access patterns*. PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1997.

I. Mandrichenko, W. Allcock, and T. Perelmutov. GridFTP v2 Protocol Description, 2005. URL http://www.ggf.org/documents/GFD.47.pdf.

J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity. *McKinsey Global Institute*, pages 1–137, 2011.

L. Massoulié and J. Roberts. Bandwidth sharing: objectives and algorithms. *IEEE/ACM Transactions on Networking (TON)*, 10(3):320–328, June 2002. ISSN 1063-6692. DOI: 10.1109/TNET.2002.1012364. URL http://dx.doi.org/10.1109/TNET.2002.1012364.

M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3):67–82, July 1997. ISSN 0146-4833. DOI: 10.1145/263932.264023. URL http://doi.acm.org/10.1145/263932.264023.

C. A. Mattmann, N. Medvidovic, P. M. Ramirez, and V. Jakobac. Unlocking the Grid. In *Proceedings of the 8th international conference on Component-Based Software Engineering*, CBSE'05, pages 322–336, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-25877-9, 978-3-540-25877-3. DOI: 10.1007/11424529_22. URL http://dx.doi.org/10.1007/11424529_22.

C. McDonald. A network specification language and execution environment for undergraduate teaching. *ACM SIGCSE Bulletin*, 23(1):25–34, Mar. 1991a. ISSN 0097-8418. DOI: 10.1145/107005.107012. URL http://doi.acm.org/10.1145/107005.107012.

C. McDonald. A network specification language and execution environment for undergraduate teaching. In *Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, SIGCSE '91, pages 25–34, New York, NY, USA, 1991b. ACM. ISBN 0-89791-377-9. DOI: 10.1145/107004.107012. URL http://doi.acm.org/10.1145/107004.107012.

Mersenne Research, Inc. Great Internet Mersenne Prime Search. Last visited, Apr. 2013. URL http://www.mersenne.org/.

MIKE2.0. Big Data definition. Last visited, Apr. 2013. URL http://mike2.openmethodology.org/wiki/Big_Data_Definition.

D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer Computing. Technical Report HPL-2002-57 (R.1), HP Laboratories Palo Alto, 2003.

M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2):226–251, 2003a. DOI: 10.1080/15427951.2004.10129088.

M. Mitzenmacher. Dynamic models for file sizes and double pareto distributions. *Internet Mathematics*, 1(3):305–333, 2003b. DOI: 10.1080/15427951.2004.10129092.

M. Mitzenmacher and B. Tworetzky. New models and methods for file size distributions. In *Proceedings of the 41st Annual Allerton Conference on Communication Control and Computing*, volume 41, number 1, pages 603–612. The University; 1998, 2003.

R. W. Moore and A. Merzky. Persistent Archive Concepts, December 2003. URL http://www.ggf.org/documents/GFD.26.pdf. No. GFD.26.

M. Morgan. ByteIO specification 1.0. Technical Report GFD-R-P.087, Open Grid Forum, Oct. 2006. URL http://www.ogf.org/documents/GFD.87.pdf.

S. A. Moyer and V. S. Sunderam. PIOUS: A scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conferece*, pages 71–78, 1994.

N. Nieuwejaar and D. Kotz. Performance of the galley parallel file system. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 83–94, 1996a.

N. Nieuwejaar and D. Kotz. The galley parallel file system. In *Proceedings of the 10th International Conference on Supercomputing*, ICS '96, pages 374–381, New York, NY, USA, 1996b. ACM. ISBN 0-89791-803-7. DOI: 10.1145/237578.237639. URL http://doi.acm.org/10.1145/237578.237639.

N. Nieuwejaar and D. Kotz. The galley parallel file system. *Parallel Computing*, 23(4âĂŞ5): 447–476, 1997. ISSN 0167-8191. DOI: 10.1016/S0167-8191(97)00009-4. URL http://www.sciencedirect.com/science/article/pii/S0167819197000094. Parallel I/O.

N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File Access Characteristics of Parallel Scientific Workloads. In *IEEE Transactions on Parallel and Distributed Systems, 7(10)*, pages 1075–1089, Oct. 1996.

Y. Niu, Z. Hu, K. Barner, and G. R. Gao. Performance modelling and optimization of memory access on cellular computer architecture cyclops64. In *Proceedings of the 2005 IFIP international conference on Network and Parallel Computing*, NPC'05, pages 132–143, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-29810-X, 978-3-540-29810-6. DOI: 10.1007/11577188_18. URL http://dx.doi.org/10.1007/11577188_18.

R. Noronha and D. Panda. Imca: A high performance caching front-end for glusterfs on infiniband. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 462–469, Sept 2008. DOI: 10.1109/ICPP.2008.84.

T. N'Takpé and F. Suter. Critical path and area based scheduling of parallel task graphs on heterogeneous platforms. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, volume 1 of *ICPADS '06*, pages 3–10, Washington, DC, USA, July 2006. IEEE Computer Society. ISBN 0-7695-2612-8. DOI: 10.1109/ICPADS.2006.32. URL http://doi.ieeecomputersociety.org/10.1109/ICPADS.2006.32.

A. Oke and R. Bunt. Hierarchical workload characterization for a busy web server. In T. Field et al., editors, *TOOLS*, pages 309–328. Springer-Verlag, Apr 2002. Lect. Notes Comput. Sci. vol. 2324.

R. Oldfield and D. Kotz. Armada: A parallel file system for computational grids. In *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 194–201. IEEE Computer Society Press, May 2001.

A. Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001. ISBN 059600110X.

T. Ott, J. Kemperman, and M. Mathis. Window size behavior in TCP/IP with constant loss probability. In *Proceedings of 4th IEEE Workshop on High-Performance Communication Systems (HPCS)*, June 1997.

Oxford University. Climateprediction.net. Last visited, Apr. 2013. URL http://climateprediction.net.

T. Özsu and P. Valduriez. *Principles of distributed database systems*. Prentice Hall, 1999. ISBN 9780136597070. URL http://books.google.es/books?id=K89QAAAAMAAJ.

J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: a simple model and its empirical validation. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '98, pages 303–314, New York, NY, USA, 1998. ACM. ISBN 1-58113-003-1. DOI: 10.1145/285237.285291. URL http://doi.acm.org/10.1145/285237.285291.

A.-M. K. Pathan and R. Buyya. A taxonomy and survey of content delivery networks. Technical Report GRIDS-TR-2007-4, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, Feb. 2007. URL http://www.cloudbus.org/reports/CDN-Taxonomy.pdf.

P. Paul. SETI@home project and its website. *Crossroads*, 8(3):3–5, 2002. ISSN 1528-4972. DOI: 10.1145/567162.567164.

D. Peleg. *Distributed computing: a locality-sensitive approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000. ISBN 0-89871-464-8.

F. Pérez, J. Carretero, F. García, P. de Miguel, and L. Alonso. Evaluating ParFiSys: a high-performance and distributed file system. *Journal of Systems Architecture. Elsevier. Vol. 43*, pages 533–542, 1997.

J. M. Pérez, L. M. Sánchez, F. García, A. Calderón, and J. Carretero. High performance Java Input/Output for heterogeneous distributed computing. *2012 IEEE Symposium on Computers and Communications (ISCC)*, 0:969–974, 2005. ISSN 1530-1346. DOI: 10.1109/ISCC.2005.79. URL http://doi.ieeecomputersociety.org/10.1109/ISCC.2005.79.

S. Phatanapherom, P. Uthayopas, and V. Kachitvichyanukul. Fast simulation model for grid scheduling using hypersim. In *Proceedings of the 2003 Winter Simulation Conference*, volume 2, pages 1494–1500 vol.2, 2003. DOI: 10.1109/WSC.2003.1261594.

P. Pierce. A concurrent file system for a highly parallel mass storage subsystem. In *Proceedings of the Fourth Conference on Hypercubes Concurrent Computers and Applications (HCCA)*, pages 155–161, 1989.

M. Placek and R. Buyya. A taxonomy of distributed storage systems. Technical Report GRIDS-TR-2006-11, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, July 2006. URL http://www.cloudbus.org/reports/DistributedStorageTaxonomy.pdf.

J. S. Plank, M. Beck, W. R. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the Network, 1999.

J. Postel and J. Reynolds. File Transfer Protocol (FTP), 1985.

R. Prodan. *Grid Computing: Experiment Management, Tool Integration, and Scientific Workflows*. Springer, Berlin, 2007. ISBN 3540692614.

X. Qin and H. Jiang. Data Grid: Supporting Data-Intensive applications in Wide-Area Networks. Technical Report TR-03-05-01, University of Nebraska-Lincoln, Lincoln, NE, May 2003.

X. Qiu, J. Ekanayake, S. Beason, T. Gunarathne, G. Fox, R. Barga, and D. Gannon. Cloud technologies for bioinformatics applications. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pages 6:1–6:10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-714-1. DOI: 10.1145/1646468.1646474. URL http://doi.acm.org/10.1145/1646468.1646474.

B. Quetier and F. Cappello. A survey of grid research tools: simulators, emulators and real life platforms. In *Proceedings of the 17th IMACS World Congress (IMACS)*, July 2005.

M. Quinson, C. Rosa, and C. Thiery. Parallel simulation of peer-to-peer systems. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 668–675, Washington, DC, USA, May 2012. IEEE Computer Society. ISBN 978-0-7695-4691-9. DOI: 10.1109/CCGrid.2012.115. URL http://hal.inria.fr/inria-00602216.

A. Rajasekar, M. Wan, R. Moore, G. Kremenek, and T. Guptil. Data grids, collections, and grid bricks. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, MSS '03, pages 2–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1914-8. URL http://dl.acm.org/citation.cfm?id=824467.825010.

K. Ranganathan and I. Foster. Design and evaluation of dynamic replication strategies for a high-performance data grid. In *International Conference on Computing in High Energy and Nuclear Physics*, 2001.

Red Hat, Inc. Gluster: Storage for your cloud. Last visited, Nov. 2015. URL http://www.gluster.org/.

W. J. Reed. The pareto, zipf and other power laws. *Economics Letters*, 74(1):15 – 19, 2001. ISSN 0165-1765. DOI: 10.1016/S0165-1765(01)00524-9. URL http://www.sciencedirect.com/science/article/pii/S0165176501005249.

W. J. Reed. The pareto law of incomes - an explanation and an extension. *Physica A: Statistical Mechanics and its Applications*, 319(0):469–486, 2003. ISSN 0378-4371. DOI: 10.1016/S0378-4371(02)01507-8. URL http://www.sciencedirect.com/science/article/pii/S0378437102015078.

W. J. Reed and B. D. Hughes. From gene families and genera to incomes and internet file sizes: Why power laws are so common in nature. *Physical Review E*, 66(6), Dec 2002. DOI: 10.1103/PhysRevE.66.067103. URL http://link.aps.org/doi/10.1103/PhysRevE.66.067103.

W. J. Reed and M. Jorgensen. The double pareto-lognormal distribution - a new parametric model for size distributions. *Communications in Statistics - Theory and Methods*, 33(8): 1733–1753, 2004. DOI: 10.1081/STA-120037438. URL http://www.tandfonline.com/doi/abs/10.1081/STA-120037438.

G. F. Riley. The Georgia Tech Network Simulator. In *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, MoMeTools '03, pages 5–12, New York, NY, USA, 2003. ACM. ISBN 1-58113-748-6. DOI: 10.1145/944773.944775. URL http://doi.acm.org/10.1145/944773.944775.

C. Roadknight, I. Marshall, and D. Vearer. File popularity characterisation. *Performance Evaluation Rev.*, 27(4):45–50, Mar 2000.

A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev.*, 35(5):188–201, 2001. ISSN 0163-5980. DOI: 10.1145/502059.502053.

L. M. Sánchez García. Diseño, implementación y evaluación del sistema de ficheros paralelo Expand 2.0. Bachelor's thesis, Computer Science department, Universidad Carlos III de Madrid, 2003.

L. M. Sánchez García. *Sistema de ficheros paralelo escalable para entornos "cluster"*. PhD thesis, Computer Science department, Universidad Carlos III de Madrid, Nov. 2009. URL http://hdl.handle.net/10016/6738.

L. M. Sánchez García, J. M. Pérez, A. Calderón, F. García Carballeira, and J. Carretero. Arquitectura escalable para E/S de altas prestaciones en sistemas heterogéneos. In *XV Jornadas de Paralelismo*, 2004.

R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the SUN network filesystem. In *Proceddings of the 1985 USENIX Conference*, pages 119–130. USENIX, 1985.

S. Sarma and E. Fleisch. Auto-ID Labs. Last visited, Oct. 2015. URL http://www.autoidlabs.org/.

M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990. ISSN 0018-9340. DOI: 10.1109/12.54838.

F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, Jan. 2002.

R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 101 –102, aug 2001. DOI: 10.1109/P2P.2001.990434.

C. Science and T. Board, editors. *Supercomputers*. National Academy Press, Washington, 1989. ISBN 0309040884.

A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, Sept. 1990. ISSN 0360-0300. DOI: 10.1145/96602.96604. URL http://doi.acm.org/10.1145/96602.96604.

H. Simitici and D. Reed. A Comparison of Logical and Physical Parallel I/O Patterns. In *International Journal of High Performance Computing Applications, special issue (I/O in Parallel Applications), 12(3)*, pages 364–380, 1998.

L. Smarr and C. E. Catlett. Metacomputing. *Commun. ACM*, 35:44–52, June 1992. ISSN 0001-0782. DOI: 10.1145/129888.129890. URL http://doi.acm.org/10.1145/129888.129890.

M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998. ISBN 0262692155.

K. Sohraby. *Wireless Sensor Networks*. Wiley, New York, 2007. ISBN 9780471743002.

S. R. Soltis, T. M. Ruwart, and M. T. OâĂŹkeefe. The global file system1. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*, volume 1, page 319. NASA, Sept. 1996.

S. R. Soltis, M. T. O'keefe, T. M. Ruwart, G. A. Houlder, J. A. Coomes, M. H. Miller, E. A. Soltis, R. W. Gilson, K. W. Preslan, et al. Global file system and data storage device locks, Dec. 2002. US Patent 6,493,804.

H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. The MicroGrid: A scientific tool for modeling computational grids. *Scientific Programming*, 8(3):127–141, Aug. 2000. ISSN 1058-9244. URL http://dl.acm.org/citation.cfm?id=1239907.1239908.

R. Stevens, P. Woodward, T. DeFanti, and C. Catlett. From the i-way to the national technology grid. *Commun. ACM*, 40:50–60, November 1997. ISSN 0001-0782. DOI: 10.1145/265684.265692. URL http://doi.acm.org/10.1145/265684.265692.

R. Subramanian and B. D. Goodman. *Peer to Peer Computing: The Evolution of a Disruptive Technology*. IGI Publishing, Hershey, PA, USA, 2005. ISBN 1591404290.

A. Sulistio, G. Poduval, R. Buyya, and C.-K. Tham. On incorporating differentiated levels of network service into gridsim. *Future Gener. Comput. Syst.*, 23(4):606–615, May 2007. ISSN 0167-739X. DOI: 10.1016/j.future.2006.10.006. URL http://dx.doi.org/10.1016/j.future.2006.10.006.

Sun Microsystems. *WebNFS Developer's Guide*, chapter 2 Extended Filesystem API. Oracle, 2004. URL http://docs.oracle.com/cd/E19455-01/806-1067/6jacl3e6g/index.html.

M. Szeredi. FUSE: Filesystem in userspace. Last visited, Apr. 2013. URL http://fuse.sourceforge.net.

A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, and U. Nagashima. Overview of a performance evaluation system for global computing scheduling algorithms. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 97–104, 1999. DOI: 10.1109/HPDC.1999.805287.

G. Tan, V. C. Sreedhar, and G. R. Gao. Analysis and performance results of computing betweenness centrality on ibm cyclops64. *J. Supercomput.*, 56:1–24, April 2011. ISSN 0920-8542. DOI: 10.1007/s11227-009-0339-9. URL http://dx.doi.org/10.1007/s11227-009-0339-9.

A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms (2nd ed.)*. Prentice Hall, 2007.

O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '02, pages 102–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1582-7. URL http://dl.acm.org/citation.cfm?id=872748.873272.

O. Tatebe, N. Soda, Y. Morita, S. Matsuoka, and S. Sekiguchi. Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing. In *Proceedings of the 14th International Conference on Computing in High Energy Physics and Nuclear Physics (CHEP'04), Interlaken, Switzerland, September 27-October 1, 2004*, pages 1172–1175, 2005. URL http://doc.cern.ch/yellowrep/2005/2005-002/p1172.pdf.

O. Tatebe, K. Hiraga, and N. Soda. Gfarm grid file system. *New Generation Computing*, 28(3): 257–275, 2010. ISSN 0288-3635. DOI: 10.1007/s00354-009-0089-5. URL http://dx.doi.org/10.1007/s00354-009-0089-5.

TechTerms.com. WWW. Last visited, Apr. 2013. URL http://www.techterms.com/definition/www.

R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, IOPADS '99, pages 23–32, New York, NY, USA, 1999a. ACM. ISBN 1-58113-123-2. DOI: 10.1145/301816.301826. URL http://doi.acm.org/10.1145/301816.301826.

R. Thakur, W. Gropp, and E. Lusk. Achieving high performance with MPI-IO. Technical Report ANL/MCS-P742-0299, Argonne National Laboratory, 1999b.

R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. Argonne National Laboratory, 1999c.

The Apache Software Foundation. HDFS Architecture Guide. Last visited, Nov. 2015a. URL https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

The Apache Software Foundation. Apache Hadoop. Last visited, Nov. 2015b. URL http://hadoop.apache.org/.

The HDF group. *HDF5 Reference Manual*, 2012. URL http://www.hdfgroup.org/HDF5/doc/RM/RM_H5Front.html.

The New York Times. Japanese 'k' computer is ranked most powerful, 2011. URL http://www.nytimes.com/2011/06/20/technology/20computer.html?_r=1.

The NS-3 Consortium. ns-3. Last visited, Apr. 2013. URL http://www.nsnam.org/.

The SimGrid Team. Simgrid. Last visited, Nov. 2014. URL http://simgrid.gforge.inria.fr/.

The University of Southern California. The network simulator - ns-2. Web page, Nov. 2011. URL http://www.isi.edu/nsnam/ns/.

S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson. Internet X.509 public key infrastructure (PKI) proxy certificate profile. Technical Report 3820, IETF, June 2004. URL http://www.ietf.org/rfc/rfc3820.txt.

P. Urbán, X. Défago, and A. Schiper. Neko: A Single Environment to Simulate and Prototype Distributed Algorithms. *Journal of Information Science and Engineering*, 18(6):981–997, 2002.

U.S. Department of Energy. *The Magellan Report On Cloud Computing for Science*. Office of Advance Scientific Computing Research (ASCR), 2011. URL http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Magellan_Final_Report.pdf.

A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, 36(SI):271–284, Dec. 2002. ISSN 0163-5980. DOI: 10.1145/844128.844154. URL http://doi.acm.org/10.1145/844128.844154.

A. Varga. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, volume 9, Prague, Czech Republic, June 2001.

A. Varga. Omnet++. In K. Wehrle, M. Güneş, and J. Gross, editors, *Modeling and Tools for Network Simulation*, pages 35–59. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-12330-6. DOI: 10.1007/978-3-642-12331-3_3.

A. Varga. The INET framework. Last visited, Apr. 2013. URL http://inet.omnetpp.org/.

A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-20-2. URL http://dl.acm.org/citation.cfm?id=1416222.1416290.

P. Velho and A. Legrand. Accuracy study and improvement of network simulation in the SimGrid framework. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 13:1–13:10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-45-5. DOI: 10.4108/ICST.SIMUTOOLS2009.5592. URL http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5592.

P. Velho, L. Schnorr, H. Casanova, and A. Legrand. Flow-level network models: have we reached the limits? Rapport de recherche RR-7821, INRIA, Nov. 2011. URL http://hal.inria.fr/hal-00646896.

S. Venugopal, R. Buyya, and K. Ramamohanarao. A taxonomy of data grids for distributed data sharing, management and processing. Technical Report GRIDS-TR-2005-3, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, Apr. 2005. URL http://www.cloudbus.org/reports/DataGridTaxonomy.pdf.

S. Venugopal, R. Buyya, and K. Ramamohanarao. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Comput. Surv.*, 38(1), June 2006. ISSN 0360-0300. DOI: 10.1145/1132952.1132955. URL http://doi.acm.org/10.1145/1132952.1132955.

W3C Technical Architecture Group. Architecture of the world wide web, volume one, 2004. URL http://www.w3.org/TR/webarch/.

F. Wang, M. Nelson, S. Oral, S. Atchley, S. Weil, B. W. Settlemyer, B. Caldwell, and J. Hill. Performance and scalability evaluation of the ceph parallel file system. In *Proceedings of the 8th Parallel Data Storage Workshop*, PDSW '13, pages 14–19, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2505-9. DOI: 10.1145/2538542.2538562. URL http://doi.acm.org/10.1145/2538542.2538562.

S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 4. IEEE Computer Society, 2004.

S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006a. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298485.

S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122. ACM, 2006b.

D. Werthimer, J. Cobb, M. Lebofsky, D. Anderson, and E. Korpela. SETI@HOME—massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1):78–83, 2001. ISSN 1521-9615. DOI: 10.1109/5992.895191.

B. White, A. S. Grimshaw, and A. Nguyen-tuong. Grid-Based File Access: The Legion I/O Model. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, pages 165–173, 2000.

B. S. White, M. Walker, M. Humphrey, and A. Grimshaw. LegionFS: A Secure and Scalable File System Support Cross-Domain High Performance Applications. In *Proceedings of Supercomputing 2001*, 2001. URL http://legion.virginia.edu/papers/SC2001.pdf.

T. White. *Hadoop: The Definitive Guide*. O'Reilly, second edition, 2011.

A. Williams, M. Arlitt, C. Williamson, and K. Barker. Web workload characterization: Ten years later. In X. Tang, J. Xu, and S. T. Chanson, editors, *Web Content Delivery*, volume 2 of *Web Information Systems Engineering and Internet Technologies Book Series*, pages 3–21. Springer Science+Business Media, Inc., 2005. ISBN 978-0-387-24356-6. DOI: 10.1007/0-387-27727-7_1.

World Wide Web Consortium (W3C). Extensible markup language (XML) 1.0, 2008. URL http://www.w3.org/TR/xml/.

Y. Yuan, Y. Wu, G. Yang, and F. Yu. Dynamic data replication based on local optimization principle in data grid. In *Proceedings of the Sixth International Conference on Grid and Cooperative Computing (GCC 2007)*, pages 815–822, 2007. DOI: 10.1109/GCC.2007.62.